7-PART SERIES · FREE COMPANION

Gift card ledger

A serverless ledger that issues gift cards and store credit, redeems them, and retires them on a schedule — with balances that are always right to the cent. Issuing mints a unique code and an opening balance; redemption is an atomic DynamoDB conditional write, so the same balance can never be spent twice from two tills at once; velocity and value caps stop a leaked code being drained; and a monthly reconciliation ties issued, redeemed, expired, and outstanding liability together. The match and the arithmetic are plain code — no model ever touches a balance. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Gift card ledger

PART 1 OF 7

JUNE 22, 2026 PART 1 OF 7 · GIFT CARD LEDGER SERIES ~9 MIN READ

A gift card ledger on AWS for a few dollars a month

A small business that sells gift cards is holding other people's money. A $50 card sold today is a $50 promise to honour later, and the moment two tills both think a card still has $50 on it, that promise quietly breaks. Spreadsheets drift, balances get spent twice, a leaked code gets drained in an afternoon, and nobody can say at month-end how much the business actually owes in unredeemed cards. This post walks through the design of a small ledger that issues a card, redeems against it atomically so a balance can never be spent twice, caps abuse, retires dormant cards on a schedule, and reconciles every cent at month-end.

KEY TAKEAWAYS

- A gift card is a promise to honour money later, so the ledger holds every balance in integer cents — never a rounded dollar figure, never a float.
- Three things create activity: issuing a card, redeeming against it, and the scheduled jobs that expire dormant cards and reconcile the books.
- The centrepiece is redemption: one atomic DynamoDB conditional write that makes a balance impossible to spend twice and impossible to push negative.
- Velocity and value caps sit on the same write, so a leaked code freezes instead of draining; an immutable ledger records every move for audit.
- Designed on AWS for about $2.10/month at small-business volume. No model ever touches a balance — the arithmetic is plain code.

The whole system on one page

Before any code, here's the shape of what we're designing. A gift card looks trivial — a code and a number that goes down — right up until two tills touch the same card at the same second, or a leaked code gets drained over a lunch break, or the owner is asked at year-end how much the business owes in unredeemed cards and the spreadsheet says one thing while reality says another. The whole system exists to make those three problems impossible rather than unlikely.

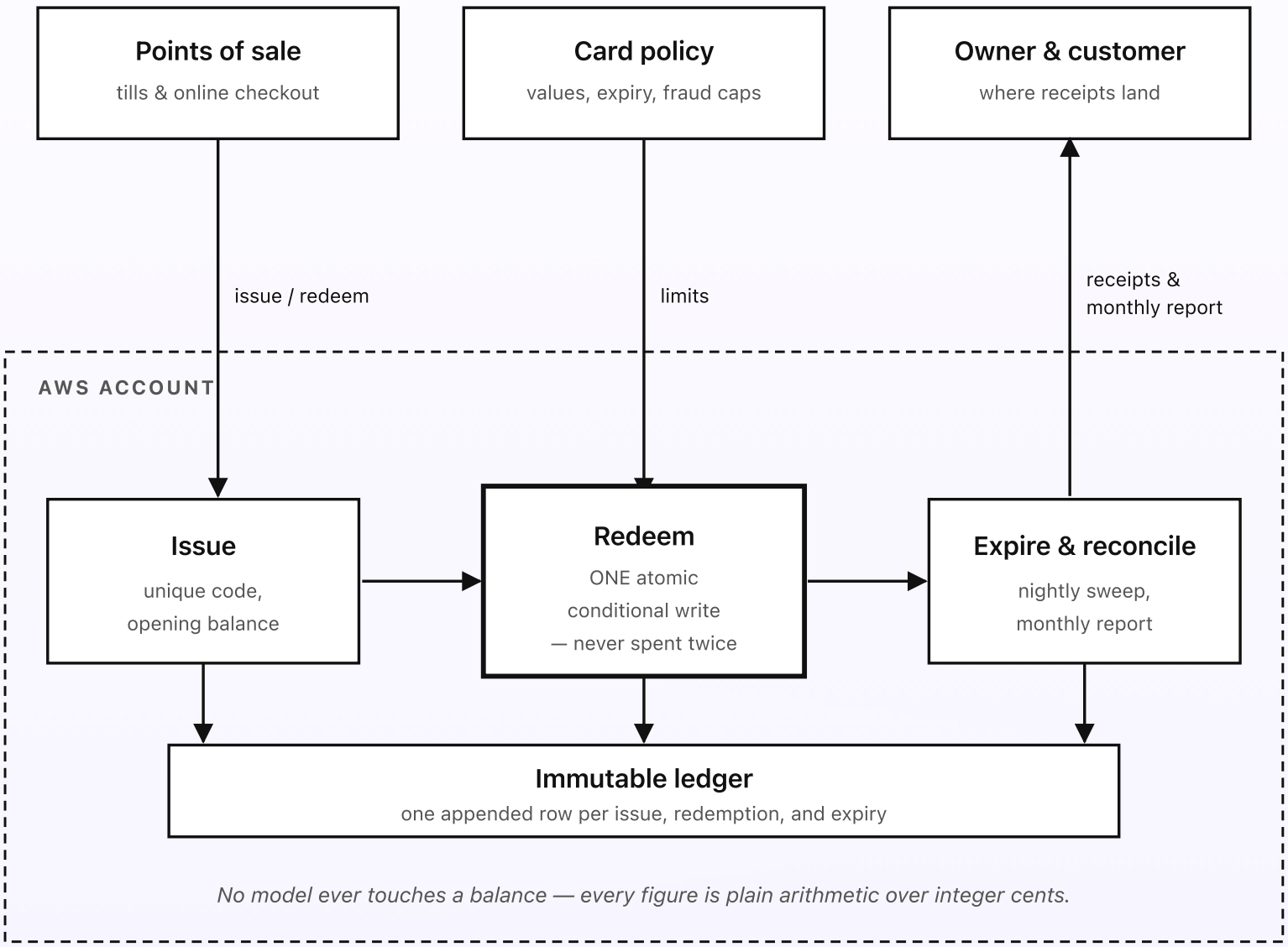

Points of sale
tills & online checkout
Card policy
values, expiry, fraud caps
Owner & customer
where receipts land
issue / redeem
limits
receipts &
monthly report
AWS ACCOUNT
Issue
unique code,
opening balance
Redeem
ONE atomic
conditional write
— never spent twice
Expire & reconcile
nightly sweep,
monthly report
Immutable ledger
one appended row per issue, redemption, and expiry
No model ever touches a balance — every figure is plain arithmetic over integer cents.

Fig 1. Activity comes from the tills and the online checkout; the policy store grounds every limit; receipts and the monthly report go back out. Inside AWS, Issue mints a card, Redeem deducts atomically, and the scheduled jobs expire and reconcile — each appending to one immutable ledger.

What you set up once

- **The card table and the ledger.** Two DynamoDB tables. The *cards* table has one row per gift card: the code, the balance in integer cents, a status (`active`, `expired`, or `void`), the date issued, the date it expires, and the timestamp of the last activity. The *ledger* table is append-only — one row for every issue, every redemption, and every expiry, never updated or deleted. The cards table gives you the live balance in one fast read; the ledger gives you the full history and the means to prove that balance is right.
- **The card policy.** A short settings store — the opening-value rules, the expiry window (a common choice is 24 months of no activity, where local law allows), and the fraud caps: the most any single redemption may take, the most a card may give up in a day, and how many redemptions are allowed in a short window before the card freezes. Changing a cap or the expiry window is a settings edit, not a deploy.
- **Receipts and the report.** SES sends the customer an issue receipt when a card is sold and a redemption receipt with the remaining balance when it is spent against. Once a month the owner gets a reconciliation: opening liability, issued, redeemed, expired, and the closing liability the business still owes.

What runs on every card

- **Issue.** Mints a unique, hard-to-guess code, sets the opening balance in cents, and writes the card with a condition that refuses to overwrite an existing one. The same step appends the opening row to the ledger. Part 2 takes this apart.
- **Redeem.** The centrepiece. A spend is a single atomic conditional write: subtract the amount *only if* the balance still covers it and no cap is tripped, and write the new balance — all in one indivisible step. If two tills hit the same card at once, exactly one succeeds and the other is told to re-read and retry. Part 3 is entirely about this, and part 4 about the caps riding alongside it.
- **Expire and reconcile.** A nightly EventBridge schedule sweeps for cards dormant past the policy window, retires them, and books the leftover as breakage. A monthly schedule re-derives every balance from the ledger and ties the books together. Part 5 covers both.

In plain words

A customer buys a $50 card at the till. Issue mints code `GC-7K2M-4QYX`, sets the balance to `5000` cents, writes the card, and emails the receipt. Three weeks later the customer spends $18.40 of it. Redeem runs one conditional write — "subtract 1840 only if the balance is still at least 1840" — the balance becomes `3160`, and the receipt says "$31.60 remaining." At the very same instant a second till tries to put through $40 on the same card; its write hits a now-too-low balance, is rejected, re-reads $31.60, and the cashier sees the real figure rather than overspending the card. Nineteen months later, if the card is never touched again, the nightly sweep retires it and books the $31.60 as breakage. None of it depended on a model, and at no point was the balance wrong.

DESIGN RULES THAT SHAPED EVERY DECISION

- Money is integer cents, everywhere. No floats, no rounded dollars, no drift.
- Redemption is one atomic conditional write. A balance can never be spent twice and can never go negative.
- Caps ride on the same write as the balance. A flagged card freezes in place rather than leaking.
- The ledger is append-only. Every move is recorded; the live balance can always be re-derived and proven.
- No model touches a balance. Bedrock writes one monthly paragraph and nothing else.
- Caps and the expiry window live in policy, not code. Tightening a limit doesn't need a deploy.

Why this shape

Most small businesses run gift cards one of two ways: a spreadsheet of codes and balances, or whatever their till software bolts on. The spreadsheet drifts the first time two people edit it, and it has no answer when a code leaks. The till add-on usually handles a single redemption fine but goes quiet on the questions that matter for money you owe — what stops the same card being spent at two tills at once, what happens when a balance is dormant for two years, and what the true outstanding liability is at month-end.

The shape above answers all three by leaning on one database guarantee: an atomic conditional write. Because the read, the check, and the write happen as a single indivisible step, there is no window for a race, so a balance is never spent twice and never goes negative — without locks, queues, or a model in the loop. Everything else hangs off that: the caps live in the same write, the ledger records every move for audit, and the monthly reconciliation re-derives every balance from that ledger to prove the books still tie out to the cent.

The next four posts walk through each piece in turn: how a gift card gets issued, how a balance gets redeemed atomically, how the fraud caps stop abuse, and how a card expires and reconciles. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 22, 2026 PART 2 OF 7 · GIFT CARD LEDGER SERIES ~6 MIN READ

How a gift card gets issued

Every gift card starts the same way: somebody pays for one, and the business now owes that amount back. Get the issue step wrong and everything downstream is built on sand — a duplicate code, a balance stored as a rounded dollar figure, or a card that exists in the till but not in the ledger. This post walks through how a card gets issued: a unique code that is awkward to guess, an opening balance held in integer cents, a conditional write that refuses to overwrite an existing card, and the first row appended to an immutable ledger.

KEY TAKEAWAYS

- A card code is a bearer instrument, so it's minted from a cryptographic random source and formatted to be awkward to guess and awkward to mistype.
- The opening balance is set once, in integer cents, from the amount actually paid — never a rounded dollar figure.
- The card is written with a strict "only if this code does not already exist" condition, so a code collision can never overwrite a live card.
- Issuing the card and appending its opening ledger row happen as one DynamoDB transaction — both land, or neither does.
- A retry of the same issue request is idempotent: it returns the existing card rather than minting a second one or charging twice.

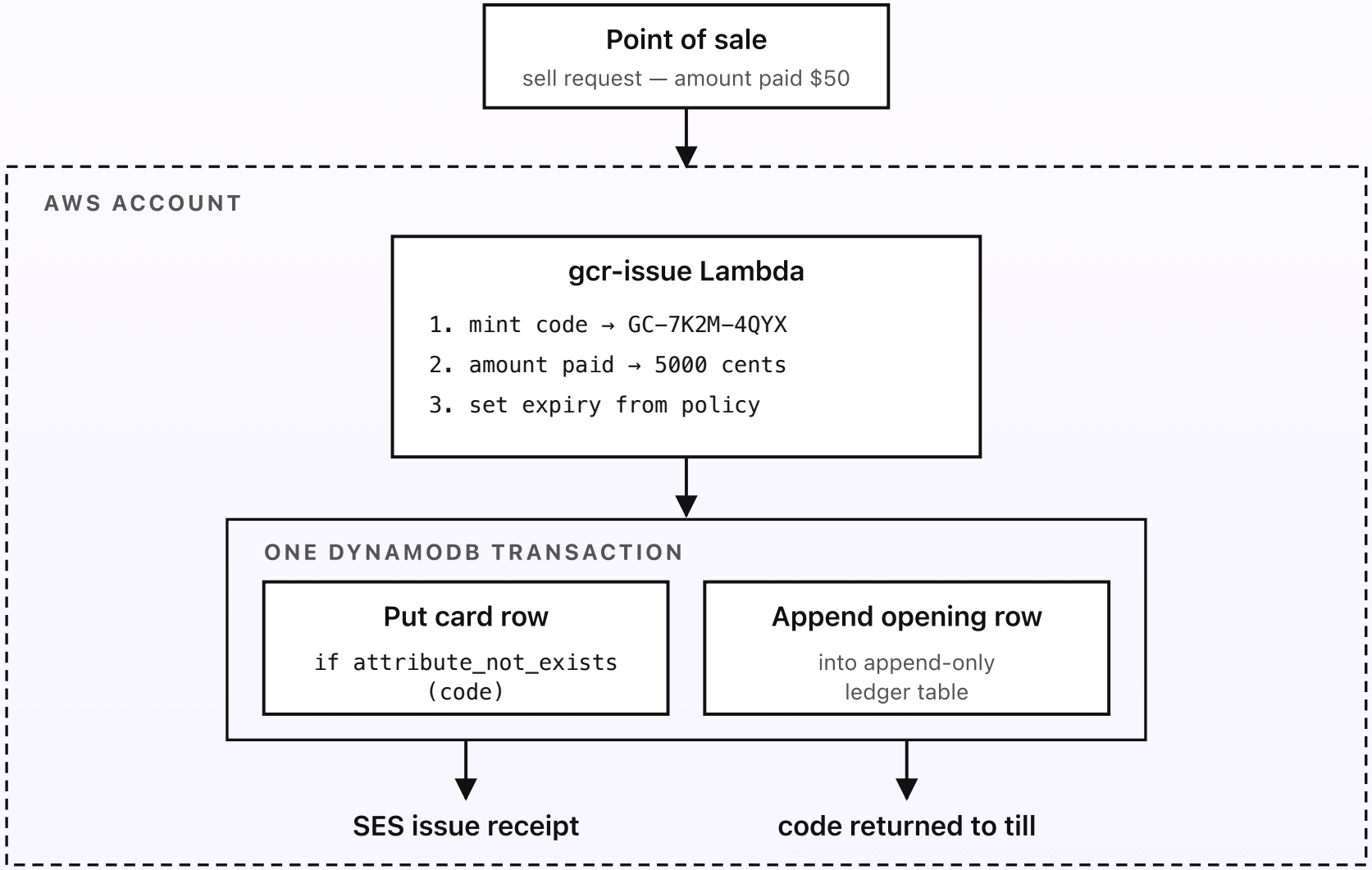
Minting a code that resists guessing

The moment a card is sold, its code becomes money. Anyone who holds the code can spend the balance, so the first job of issuing is to mint a code that is hard to guess from another one and hard to stumble onto at random. A sequential number — card 1001, card 1002 — is the worst case: knowing one card tells you a hundred others. Instead the issue step pulls bytes from a cryptographic random source and encodes them into a code like `GC-7K2M-4QYX`: a fixed prefix so staff recognise it, then two groups drawn from an alphabet that drops the easily-confused characters (no `0` / `O`, no `1` / `I` / `l`) so it survives being read aloud or typed off a printed slip.

The code space is large enough that guessing is hopeless — far more combinations than a business will ever issue — so an attacker cannot simply try codes until one has a balance. That property matters again in part 4, where the fraud caps assume a leaked code is the realistic threat, not a guessed one.

Setting the balance in cents, once

The opening balance is whatever the customer actually paid, stored as an integer number of cents. A $50 card is `5000`, a $25 card is `2500`, a $37.50 promotional card is `3750`. There is no floating-point anywhere near a balance: `0.1 + 0.2` famously is not `0.3` in a float, and a ledger that holds other people's money cannot afford that drift accumulating over thousands of redemptions. Every figure in the system — balances, spends, caps, breakage — is an integer count of cents, and only ever formatted as dollars for display on a receipt.



A repeat of the same request returns the existing card — it never mints a second or charges twice.

Fig 2. The issue step mints a code, converts the amount paid to cents, and sets the expiry, then writes the card row and the opening ledger row in one transaction guarded by `attribute_not_exists(code)`. A repeat request is idempotent.

Writing the card without clobbering one

Two writes have to land together: the card itself in the cards table, and its opening row in the ledger. The issue step does them as a single DynamoDB transaction, so either both commit or neither does — there is no state where a card exists with no ledger history, or a ledger row points at a card that was never written.

The card write carries a condition: `attribute_not_exists(code)`. It means "create this card only if no card with this code already exists." The code space is enormous, so a genuine collision is astronomically unlikely — but "astronomically unlikely" is not "impossible", and a system holding money should never silently overwrite a live balance. If the condition fails, the issue step simply mints a fresh code and tries again, and nobody's balance is touched.

Making a retry safe

Tills lose connectivity mid-sale. A cashier taps "sell", the network drops before the response gets back, and the natural reaction is to tap it again. Without care that mints two cards and, worse, could charge the customer twice. The issue step is therefore idempotent: every sell request carries a client-supplied request id, and the step records that id against the card it created. A repeat with the same id returns the card that already exists — same code, same balance — rather than

minting a second one. The till sees one card, the customer is charged once, and the ledger has exactly one opening row.

DESIGN RULES FOR ISSUING

- Codes come from a cryptographic random source, not a sequence, and skip easily-confused characters.
- The opening balance is integer cents taken from the amount actually paid.
- The card row and its ledger row are written in one transaction — both or neither.
- A strict `attribute_not_exists(code)` condition means a code can never overwrite a live card.
- A repeat sell request is idempotent by request id: one card, one charge, one opening row.

PART 3 OF 7

JUNE 22, 2026 PART 3 OF 7 · GIFT CARD LEDGER SERIES ~7 MIN READ

How a balance gets redeemed

This is the post the whole series is built around. A customer hands over a gift card; a till needs to take $18.40 off it and be certain — even if another till is touching the same card at the same instant — that the figure that comes out is exactly right. The naive way reads the balance, subtracts, and writes it back, and that gap between read and write is precisely where money goes missing. This post walks through how a balance gets redeemed with one atomic conditional write that closes the gap entirely: no double-spend, no negative balance, no model in the loop.

KEY TAKEAWAYS

- The naive read-subtract-write has a gap between reading the balance and writing it back — and that gap is exactly where a balance gets spent twice.
- A redemption is instead one atomic DynamoDB `UpdateItem`: check, subtract, and write in a single indivisible step.
- A condition expression of `balance >= :amount` means a balance can never go negative and can never be overspent.
- If two tills hit the same card at once, DynamoDB applies exactly one and rejects the other with a `ConditionalCheckFailedException`.
- The rejected till re-reads the now-lower balance and retries — no locks, no queue, no model, and the figure is always right to the cent.

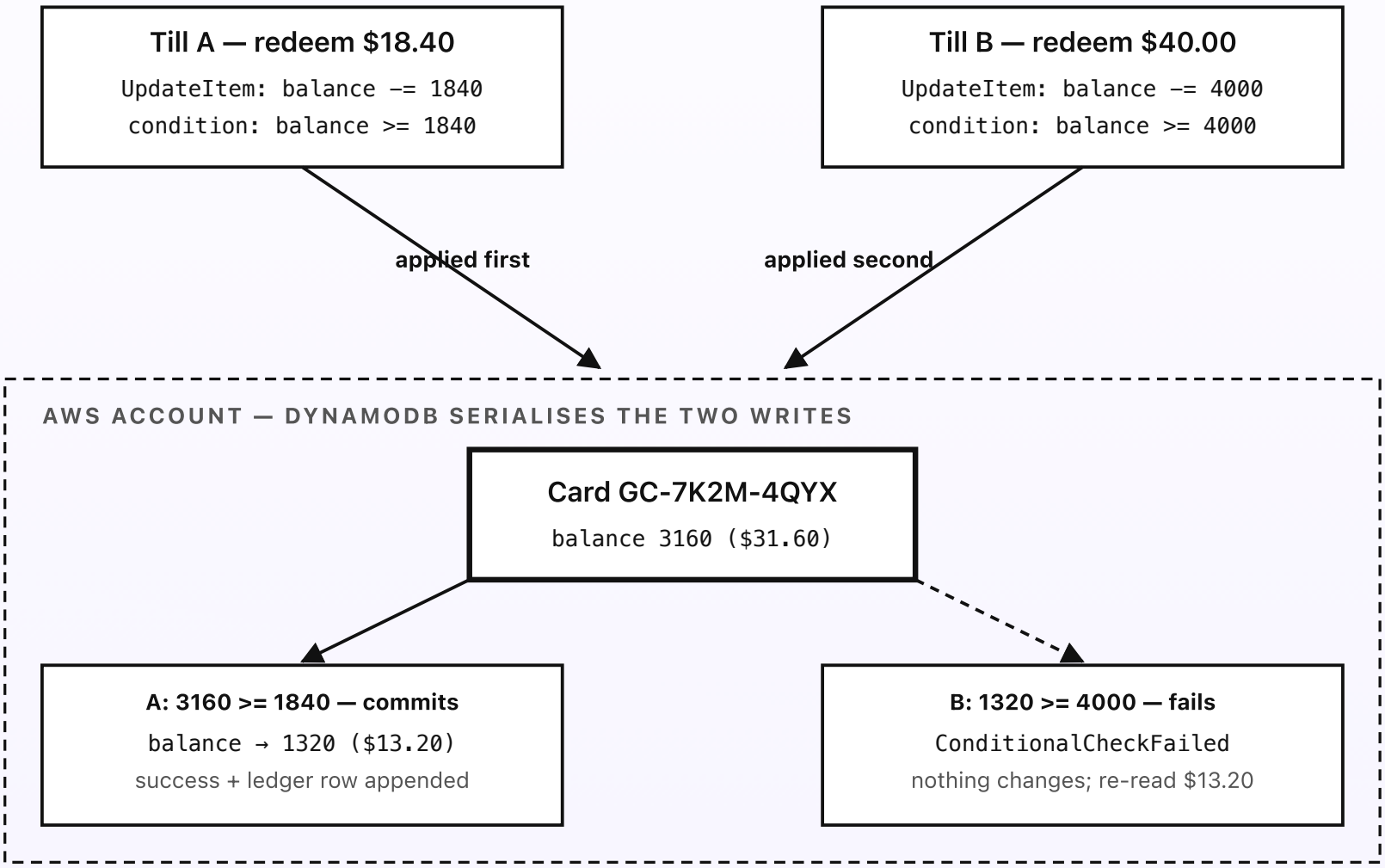
The gap where money goes missing

Picture the obvious way to spend $18.40 off a card: read the balance ($50.00), subtract in code ($31.60), write the new balance back. On one till, on a quiet afternoon, this works perfectly. The trouble is two tills and the same card at the same second — a customer paying at the counter while a colleague applies the same card to an online order, say. Till A reads $50.00. A heartbeat later, before A has written anything, till B also reads $50.00. A writes $31.60. B, working from the $50.00 it read, writes its own figure and clobbers A's. Two redemptions happened; only one came off the balance. The business just gave away $18.40.

That gap between the read and the write is the whole problem. Any design that reads a balance, thinks about it, and writes it back later has the gap, no matter how short. Locks can close it, but locks bring their own trouble — a till that takes a lock and then loses its connection leaves a card stuck until something times out. The cleaner answer is to never have the gap at all.

One atomic conditional write

DynamoDB lets you do the check, the subtraction, and the write as a single indivisible operation. A redemption is one `UpdateItem` call that says, in effect: *subtract this amount from the balance, but only if the balance is still at least this amount*. In the API that is an `UpdateExpression` of `SET balance = balance - :amount` guarded by a `ConditionExpression` of `balance >= :amount`. DynamoDB evaluates the condition and applies the change atomically — nothing can read or write that card's balance in between, because there is no "in between."



No locks, no queue, no model — a balance can never be spent twice and can never go negative.

Fig 3. Two tills race for the same $31.60 card. DynamoDB serialises the writes: Till A's condition holds and commits ($13.20 left); Till B's condition now fails and is rejected. Till B re-reads the real balance and the card is never overspent.

What happens in a race

Take the same two tills from the diagram. The card holds $31.60 (`3160` cents). Till A wants $18.40 (`1840`); till B wants $40.00 (`4000`). Both fire at once. DynamoDB cannot apply two writes to the same item at literally the same moment — it serialises them, picking one to go first.

Say A goes first. Its condition, `3160 >= 1840`, holds, so the subtraction commits and the balance becomes `1320`. Till A gets a success and appends its ledger row. Now B is evaluated — but against the new balance, `1320`. Its condition, `1320 >= 4000`, fails, so DynamoDB rejects the write with a `ConditionalCheckFailedException` and changes nothing. No partial spend, no negative balance, no clobbering of A's write. The order could just as easily have been B then A; the guarantee is the same either way — whatever the balance can actually cover gets spent, and no more.

Rejection is a normal outcome, not an error

A rejected redemption is not a failure to log and panic over — it is the system working. The redeem step catches the `ConditionalCheckFailedException` and treats it as a clear, expected signal: this spend did not fit the current balance. It re-reads the card, sees `1320`, and hands the till the truth: "$13.20 remaining." The cashier can then take $13.20 off the card and the rest on another tender,

instead of overspending it. If the rejection was caused purely by a race — the amount *would* have fit, but another write got there first — the step simply retries against the fresh balance, with a couple of quick attempts and a little jitter so two racing tills do not lock-step.

One more detail keeps the books clean: the ledger row is appended only when the conditional write actually commits. A rejected attempt writes nothing to the ledger, so the history shows exactly the redemptions that happened and nothing else. And, as everywhere in this system, not a line of this depends on a model — it is plain arithmetic over integer cents, decided by a database condition.

DESIGN RULES FOR REDEMPTION

- One atomic `UpdateItem` per redemption — check, subtract, and write in a single step. Never read-then-write.
- The condition `balance >= :amount` guarantees no overspend and no negative balance.
- A `ConditionalCheckFailedException` is an expected outcome: re-read and either retry or tell the till the real balance.
- The ledger row is appended only on a committed write, so the history matches reality exactly.
- No locks, no queue, no model — the database guarantee does the work.

PART 4 OF 7

JUNE 22, 2026 PART 4 OF 7 · GIFT CARD LEDGER SERIES ~8 MIN READ

How fraud caps stop abuse

A gift card code is a bearer instrument: whoever holds it can spend it. So when a code leaks — photographed, forwarded, scraped from an inbox — the only thing standing between a thief and the full balance is how fast they can spend it. This post walks through the fraud caps that sit on top of the atomic redemption: a ceiling on any single redemption, a daily limit per card, and a velocity check on how many redemptions happen in a short window. Each is enforced in the same indivisible write as the balance, so a card that trips a cap freezes on the spot instead of draining.

KEY TAKEAWAYS

- A leaked code — not a guessed one — is the realistic threat, and the caps are built to blunt exactly that.
- Three caps: a per-redemption ceiling, a daily limit per card, and a velocity limit on how many redemptions fit in a short window.
- Every cap is enforced inside the same atomic conditional write as the balance, so a tripped cap never spends first and asks later.
- The card carries a rolling daily total and a recent-count, both reset by time, so the checks are a single self-contained write.
- A card that trips a cap freezes — it's flagged for review rather than drained — and every freeze lands in the ledger and an alert.

The threat is a leaked code, not a guessed one

Part 2 made the code space large enough that guessing a live card is hopeless. So the realistic way a card gets abused is not guessing — it is a leak. A photo of the card posted online, a forwarded confirmation email, a code skimmed from a printed slip, a batch of e-gift codes scraped from a compromised inbox. Once a code is loose, whoever holds it can spend the balance, and the only thing that limits the damage is how fast and how freely they can spend before anyone notices.

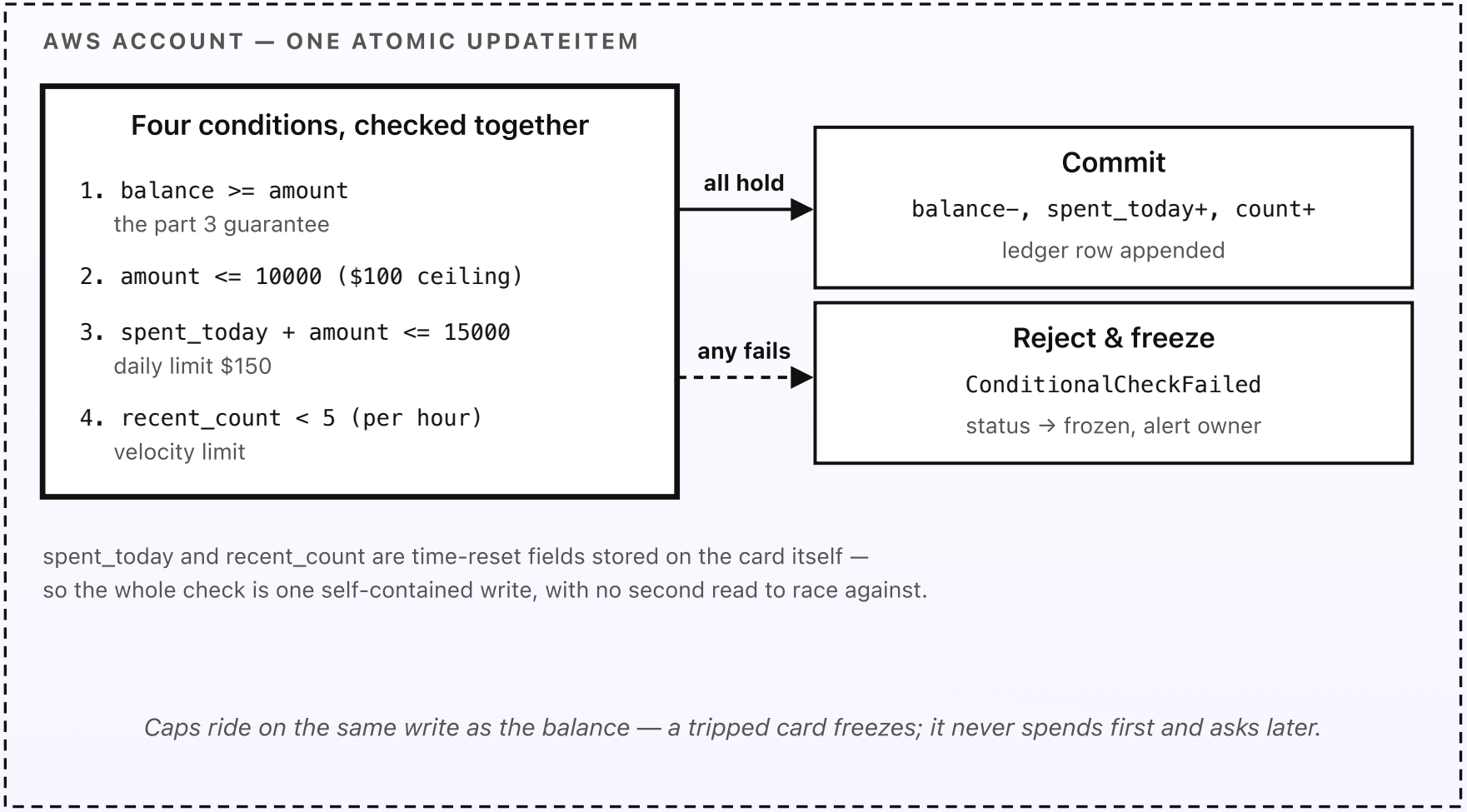
The caps exist to make a leaked code worth far less than the balance on it. They cannot stop the first small spend — that is indistinguishable from a real customer

— but they can stop a $200 card being emptied in three rapid hits from a script, and they can stop a single redemption walking off with the whole balance at once.

Three caps, all on the hot path

- **A per-redemption ceiling.** No single redemption may take more than a set amount — say $100, or the opening value, whichever the policy sets. This stops one request draining a large balance in a single move and makes a stolen high-value card behave like a small one.
- **A daily limit per card.** A card may give up at most a set amount per calendar day — say $150. A genuine customer rarely bumps into this; a thief trying to liquidate a card before it's reported hits it fast.
- **A velocity limit.** At most a set number of redemptions — say 5 — in a rolling short window such as an hour. Real spending is bursty but small in count; a script hammering a code is not. The velocity cap is the one that catches automated draining.

The important design choice is *where* these run. It would be easy to check the caps first, in code, and then do the redemption — but that reopens exactly the gap part 3 spent its whole length closing. Two requests could both pass the cap check and then both spend. So the caps do not run before the write; they run *inside* it.



spent_today and recent_count are time-reset fields stored on the card itself —
so the whole check is one self-contained write, with no second read to race against.

Caps ride on the same write as the balance — a tripped card freezes; it never spends first and asks later.

Fig 4. A redemption is one atomic write guarded by four conditions: balance, the per-redemption ceiling, the daily limit, and the velocity limit. All four hold and it commits; any one fails and the write is rejected and the card frozen.

Caps inside the same write

The redeem step keeps two extra fields on the card row alongside the balance: `spent_today`, a rolling total of cents redeemed so far today, and `recent_count`, how many redemptions have landed in the current short window. Both carry the timestamp of their window, so the redeem step knows when to treat them as reset to zero rather than carrying yesterday's figure forward.

Because all three caps read off fields on the same card row as the balance, every check folds into one condition expression on a single `UpdateItem`. The write commits only if *all* of these hold at once: `balance >= :amount`, `:amount <= :ceiling`, `spent_today + :amount <= :daily`, and `recent_count < :velocity`. When it commits, the same write subtracts the balance, adds to `spent_today`, and increments `recent_count` — so the counters can never drift out of step with the spend. There is no separate read of the counters and therefore no gap for two requests to both slip through. The caps inherit the exact guarantee from part 3.

What a tripped cap does

When the condition fails, DynamoDB rejects the write and nothing on the card changes — no balance moves, no counter ticks. The redeem step then has to decide why it failed. A plain "balance too low" is the ordinary part 3 case: tell the

till the real balance. But a cap failure — especially the velocity cap — is treated as suspicious. The card's status is moved to `frozen`, which makes every future redemption fail closed until a human clears it, an alert goes to the owner with the code and what tripped it, and the attempt is recorded in the ledger as a frozen event. A real customer who genuinely hit a daily limit can be unfrozen with a quick look; a thief gets a card that stops paying out after a few cents rather than after a few hundred dollars.

Freezing rather than silently declining matters. A silent decline tells an attacker probing a code exactly where the limit sits, so they tune just under it. A freeze takes the card out of play entirely on the first trip and puts a person in the loop — which is the right place for a judgement call about someone else's money.

DESIGN RULES FOR FRAUD CAPS

- Assume the threat is a leaked code; size the caps to make a leaked card worth little.
- Three caps: per-redemption ceiling, daily limit, velocity limit — all in policy, not code.
- Every cap is a condition on the same atomic write as the balance. Never check first, spend second.
- Counters live on the card row and update in the same write, so they can't drift from the spend.
- A tripped cap freezes the card and alerts a human — it fails closed, never silently declines.

PART 5 OF 7

JUNE 22, 2026 PART 5 OF 7 · GIFT CARD LEDGER SERIES ~7 MIN READ

How a card expires and reconciles

Two slow, unglamorous jobs keep a gift card ledger honest over time. The first is expiry: cards that have sat untouched past the policy window need retiring, and the leftover balance booked as breakage, cleanly and on a schedule rather than by hand. The second is reconciliation: at month-end the business needs one trustworthy number for how much it still owes in live cards. This post walks through the nightly expiry sweep and the monthly reconciliation that re-derives every balance from the immutable ledger and proves the books still add up to the cent.

KEY TAKEAWAYS

- A nightly EventBridge schedule sweeps for cards dormant past the policy window and retires them, booking the leftover balance as breakage.
- Expiry is itself an atomic conditional write, so a card being spent at the same instant is never wrongly expired.
- A monthly schedule re-derives every balance from the immutable ledger and checks it against the stored figure.
- The reconciliation proves one identity: opening liability + issued − redeemed − expired = closing liability, to the cent.
- Bedrock writes a short plain-English narrative on top of the finished numbers — it never computes or touches a figure.

Retiring a dormant card

Gift cards do not get spent down to zero and closed off neatly. A good fraction are partly used and then forgotten — the $50 card with $31.60 still on it that nobody ever brings back. Left alone, those balances sit on the books as money owed forever, and the outstanding-liability figure slowly loses meaning. Where local law allows it, the policy sets an expiry window — commonly 24 months with no activity — after which a dormant card is retired and its leftover balance is recognised as *breakage*: value that was sold but will never be redeemed.

A nightly EventBridge schedule runs the expiry sweep at a quiet hour. It queries for cards whose last-activity timestamp is older than the window and whose status is still `active`, and retires each one. Crucially, retiring a card is not a blind overwrite — it is an atomic conditional write, the same tool as everywhere else. The sweep says "set this card to `expired` and book the balance as breakage, *only if* the last-activity timestamp is still the old one I read." If the customer happened to spend the card a second before the sweep touched it, the timestamp has moved, the condition fails, and the card is left active — their money is safe. No race between a redemption and the sweep can ever expire a card that was just used.

Each expiry appends a row to the ledger — an `expired` event with the breakage amount — so the history stays complete and the monthly reconciliation can account for every cent that left the live pool.

Proving the books once a month

The balance on a card row is fast to read, but "fast" is not "trusted." The trusted number is the one you can rebuild from scratch. Once a month a second EventBridge schedule runs the reconciliation, which re-derives every card's balance from the immutable ledger — opening row plus every redemption minus, every expiry out — and checks the rebuilt figure against the balance stored on the card row. If a single card disagrees, the report names it rather than papering over it. In normal running they always agree, because every write that moved a balance also appended its ledger row in the same step.

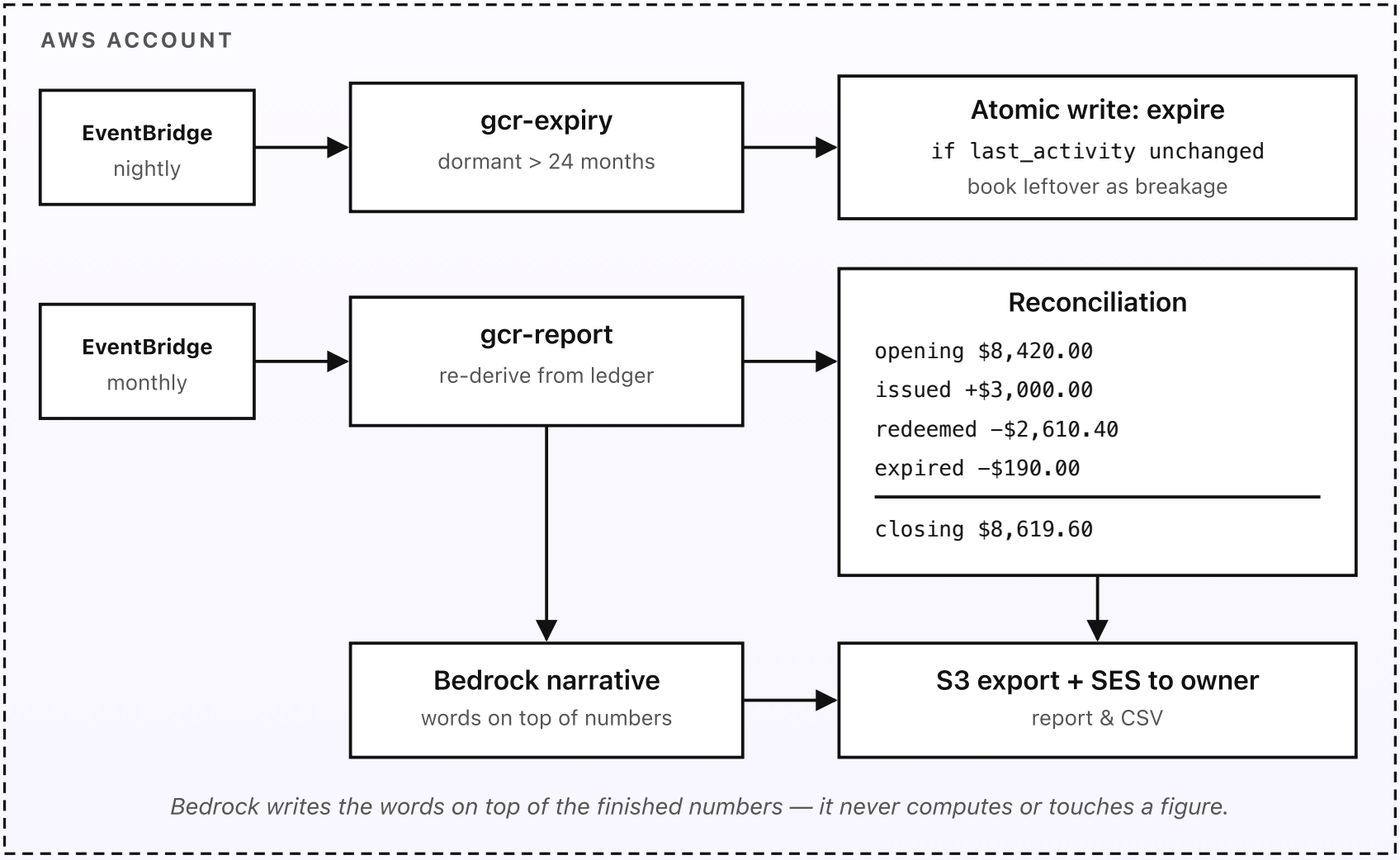

AWS ACCOUNT
EventBridge
nightly
gcr-expiry
dormant > 24 months
Atomic write: expire
if last_activity unchanged
book leftover as breakage
EventBridge
monthly
gcr-report
re-derive from ledger
Reconciliation
opening $8,420.00
issued +$3,000.00
redeemed −$2,610.40
expired −$190.00
closing $8,619.60
Bedrock narrative
words on top of numbers
S3 export + SES to owner
report & CSV
Bedrock writes the words on top of the finished numbers — it never computes or touches a figure.

Fig 5. Two schedules keep the ledger honest: a nightly sweep expires dormant cards with an atomic conditional write, and a monthly report re-derives every balance from the ledger and ties opening, issued, redeemed, expired, and closing liability together.

The one identity that must hold

The whole reconciliation comes down to a single equation that has to balance to the cent:

```
opening liability + issued − redeemed − expired = closing liability
```

Each term is summed straight from the ledger over the month. *Opening* is last month's closing figure. *Issued* is every opening row — the new money the business took on. *Redeemed* is every redemption row — money handed back as goods. *Expired* is every breakage row — money recognised as never coming back. *Closing* is what the business still owes in live cards. A worked month: opening $8,420.00, issued $3,000.00, redeemed $2,610.40, expired $190.00, and so closing $8,619.60. If the rebuilt closing figure does not equal the sum of the live card balances, the report flags the gap loudly — that mismatch is the single most important alarm in the system, because it means a balance moved without a matching ledger row, and the books can no longer be trusted until it is explained.

Words on top of numbers

By the time the report is assembled, every figure is final and was computed in plain Python over integer cents. The last step hands those finished numbers to a

single Bedrock Haiku call that writes a short, readable narrative for the owner: how many cards were issued and redeemed, how much was recognised as breakage, how the outstanding liability moved, and anything the report flagged. The model only ever phrases numbers it is given; it never adds them up, never reconciles, and never touches a balance. Switch Bedrock off and every figure in the report is identical — you simply lose the paragraph of prose. The report and a CSV export are written to S3 (versioned, so each month's close is preserved) and emailed to the owner through SES.

DESIGN RULES FOR EXPIRY AND RECONCILIATION

- Expiry is a scheduled job, and each retirement is an atomic conditional write guarded on last-activity — a just-spent card is never wrongly expired.
- Breakage is recognised explicitly and appended to the ledger, never quietly dropped.
- The reconciliation re-derives every balance from the immutable ledger, not from the live row it is checking.
- One identity must hold to the cent: opening + issued − redeemed − expired = closing. A mismatch is the loudest alarm in the system.
- Bedrock writes the narrative on top of finished numbers — it never computes a figure.

PART 6 OF 7

JUNE 22, 2026 PART 6 OF 7 · GIFT CARD LEDGER SERIES ~5 MIN READ

What the gift card ledger costs

A ledger that holds other people's money should not cost much to run, and this one does not. This post breaks the bill down line by line at a realistic small-business volume — around 300 cards issued and 800 redemptions a month — and shows where every cent of the roughly $2.10 monthly total goes. The headline is that the centrepiece, the atomic redemption, costs a fraction of a cent each; the bill is dominated by one Secrets Manager secret and keeping a continuous backup of the ledger. We finish with the bill at ten times the volume.

KEY TAKEAWAYS

- About $2.10/month at a realistic volume of roughly 300 cards issued and 800 redemptions.
- The centrepiece, the atomic redemption, is a single conditional write — a fraction of a cent each, a few cents for the whole month.
- The bill is dominated by fixed costs: one Secrets Manager secret and a continuous backup of the ledger.
- EventBridge Scheduler, SQS, and AWS Budgets all sit inside their free tiers and add nothing.
- At ten times the volume the bill lands around $9/month, because most of it scales with writes and email, not with a server.

The volume we're pricing

Costs only mean something against a volume, so here is the one we're pricing: a small business that issues about **300 gift cards a month** and takes about **800 redemptions** against the live pool. The nightly expiry sweep runs once a day; the reconciliation report runs once a month. Receipts go out on issue and on redemption, so SES sends on the order of 1,100 emails a month plus the one report. Everything lives in a single region, `eu-west-2`, with no always-on compute anywhere.

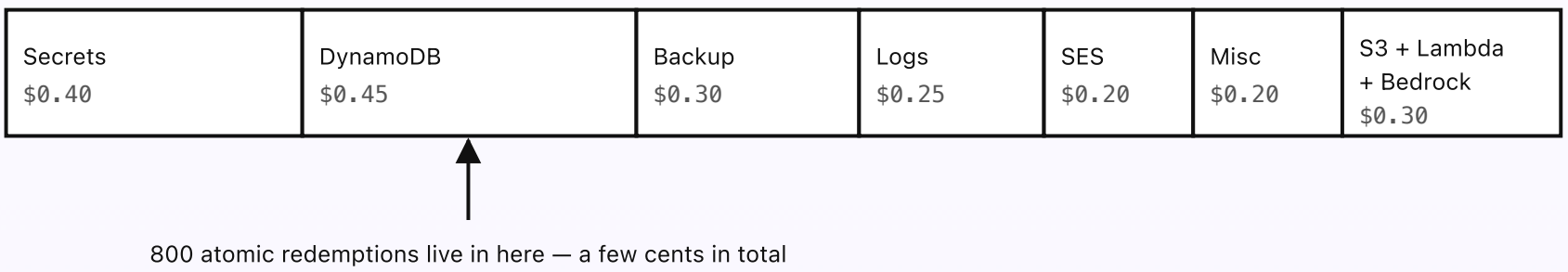
The line-by-line bill

Service	What it does here	Monthly
Secrets Manager	One secret — the SES / config credentials	$0.40
DynamoDB (on-demand)	Writes & reads: ~300 issues, ~800 redemptions, sweep + report scans	$0.45
DynamoDB backup (PITR)	Continuous point-in-time recovery on the cards & ledger tables	$0.30
CloudWatch Logs	Function logs at 7-day retention	$0.25
SES	~1,100 issue & redemption receipts + the monthly report	$0.20
Data transfer & misc	Small request and egress overheads	$0.20
S3 (versioned)	Monthly reports, CSV exports, ledger snapshots	$0.15
Lambda (Python 3.14, arm64)	Issue, redeem, expiry, report invocations	$0.10
Bedrock (Claude Haiku 4.5)	One report-narrative call a month	$0.05
EventBridge Scheduler	Nightly sweep + monthly report (free tier)	$0.00

Service	What it does here	Monthly
SQS + dead-letter queue	Receipt / alert buffering (free tier)	$0.00
AWS Budgets	Two budgets and an alert (free tier)	$0.00
Total		**$2.10**

Where the money actually goes

The striking thing about this bill is that the part everyone worries about — getting redemptions right under load — is the cheapest line on it. Each redemption is one DynamoDB conditional write; 800 of them, plus the issues and the scheduled scans, come to well under half a dollar for the month. The expensive lines are fixed and have nothing to do with how busy the shop is: one Secrets Manager secret at $0.40, and continuous backup of the cards and ledger tables at $0.30. For a system holding other people's money, paying $0.30 a month to be able to rewind the ledger to any second is the easiest line item to justify on the whole list.



Free tier, $0.00: EventBridge Scheduler · SQS + DLQ · AWS Budgets.

The two fixed lines — the secret and the backup — are the biggest combined share of the bill.

Fig 6. The $2.10 month split by service. The 800 atomic redemptions sit inside the small DynamoDB writes segment; the largest combined share is fixed cost — one secret and a continuous ledger backup — not anything that scales with the shop.

What ten times the volume costs

Now run the same shape at ten times the traffic: about **3,000 cards issued** and **8,000 redemptions** a month. The fixed lines do not move — the secret is still

$0.40, the report still fires one Bedrock call — while the usage-based lines scale roughly with the work. DynamoDB writes and reads grow toward $3.50–$4.00, SES toward $1.50, logs and data transfer up in step, and continuous backup edges up with the ledger's size. The total lands around **$9 a month**. Even at ten times the customers, there is no server to grow, no cluster to size, and no idle capacity billed overnight — the bill tracks the number of cards and emails, and almost nothing else.

WHY THE BILL STAYS SMALL

- No always-on compute. Lambda bills per invocation; between sales the system costs nothing to run.
- The atomic redemption — the hard part — is one conditional write, the cheapest line on the bill.
- On-demand DynamoDB means you pay per request, not for provisioned throughput sitting idle.
- The fixed floor is small and honest: one secret and a continuous ledger backup.
- Scheduler, SQS, and Budgets stay inside the free tier at this volume and add nothing.

PART 7 OF 7

JUNE 22, 2026 PART 7 OF 7 · GIFT CARD LEDGER SERIES ~7 MIN READ

Engineering reference: the gift card ledger architecture

This is the reference post: the gift card ledger drawn for someone who is going to build it. No shop-floor framing — just the resources and how they fit. The Lambda inventory with the gcr- prefix, the two EventBridge schedules, the DynamoDB tables with their partition and sort keys, the precise conditional expression behind an atomic redemption and the transaction behind an issue, the S3 layout, the SES senders, the IAM scopes, and the one region everything lives in. Read part 3 first if you want the why; this is the what.

KEY TAKEAWAYS

- Four Lambdas (`gcr-issue`, `gcr-redeem`, `gcr-expiry`, `gcr-report`), all Python 3.14 on arm64, in one region.
- Two DynamoDB tables on-demand: `gc-cards` (live state) and `gc-ledger` (append-only history), both with PITR.
- The atomic redemption is one `UpdateItem` with a four-clause condition; the issue is a two-write `TransactWriteItems`.
- Two EventBridge schedules drive the nightly expiry sweep and the monthly reconciliation report.
- IAM is scoped per function to the exact tables and actions it needs; everything lives in `eu-west-2`.

The architecture, for engineers

Same system as the rest of the series, drawn without the shop-floor framing. Every resource carries the `gcr-` prefix; everything sits in one region.

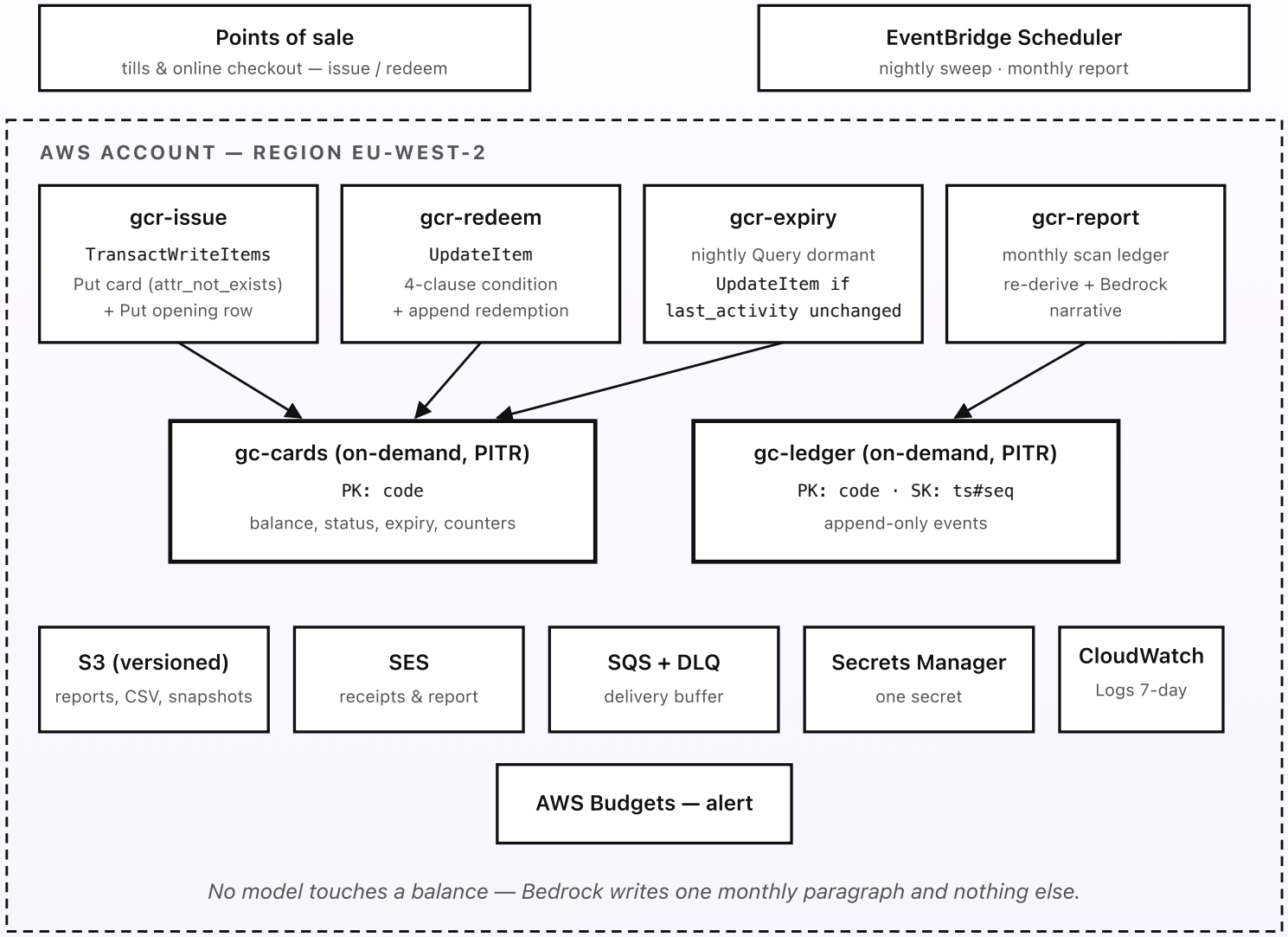

Points of sale
tills & online checkout — issue / redeem
EventBridge Scheduler
nightly sweep · monthly report
AWS ACCOUNT — REGION EU-WEST-2
gcr-issue
TransactWriteItems
Put card (attr_not_exists)
+ Put opening row
gcr-redeem
UpdateItem
4-clause condition
+ append redemption
gcr-expiry
nightly Query dormant
UpdateItem if
last_activity unchanged
gcr-report
monthly scan ledger
re-derive + Bedrock
narrative
gc-cards (on-demand, PITR)
PK: code
balance, status, expiry, counters
gc-ledger (on-demand, PITR)
PK: code · SK: ts#seq
append-only events
S3 (versioned)
reports, CSV, snapshots
SES
receipts & report
SQS + DLQ
delivery buffer
Secrets Manager
one secret
CloudWatch
Logs 7-day
AWS Budgets — alert
No model touches a balance — Bedrock writes one monthly paragraph and nothing else.

Fig 7. The full ledger in one region: four Lambdas over two on-demand DynamoDB tables, with the atomic redemption as a single conditional `UpdateItem` and the issue as a two-write transaction. S3, SES, SQS, Secrets Manager, CloudWatch, and Budgets support the edges.

Lambda inventory

- **`gcr-issue`** — Python 3.14, arm64. Mints a code from a CSPRNG, sets the opening balance in cents and the expiry date, and commits a `TransactWriteItems` of two operations: a `Put` into `gc-cards` with `ConditionExpression: attribute_not_exists(code)`, and a `Put` of the opening row into `gc-ledger`. Idempotent on a client request id.
- **`gcr-redeem`** — Python 3.14, arm64. The hot path. One `UpdateItem` on `gc-cards` carrying the four-clause condition and the counter updates; on success it appends a redemption row to `gc-ledger`. Catches `ConditionalCheckFailedException` and either retries (race) or returns the real balance (insufficient), and freezes the card on a cap trip.
- **`gcr-expiry`** — Python 3.14, arm64. Triggered nightly. Queries `gc-cards` for cards past the dormancy window, then issues conditional `UpdateItem` calls guarded on `last_activity` to set `status = expired` and book breakage, appending an `expired` row to `gc-ledger` for each.
- **`gcr-report`** — Python 3.14, arm64. Triggered monthly. Scans `gc-ledger` for the period, re-derives every balance, checks the reconciliation identity, calls Bedrock Haiku 4.5 for the narrative, and writes the report and CSV to S3 and emails the owner via SES.

Schedules

- **Nightly expiry sweep** — EventBridge Scheduler, a fixed off-peak time in `eu-west-2`, invoking `gcr-expiry` once a day.
- **Monthly reconciliation** — EventBridge Scheduler, first of the month, invoking `gcr-report` once.

DynamoDB tables and key schema

- **`gc-cards`** — on-demand, PITR on. Partition key `code` (string), no sort key — one item per card. Attributes: `balance` (number, integer cents), `status` (`active` / `expired` / `void` / `frozen`), `issued_at`, `expires_at`, `last_activity`, `spent_today`, `today` (date stamp), `recent_count`, `window_start`, and the issue `request_id`. The redemption is a single `UpdateItem` with `UpdateExpression: SET balance = balance - :amt, spent_today = ..., recent_count = ...` and `ConditionExpression: balance >= :amt AND :amt <= :ceiling AND spent_today + :amt <= :daily AND recent_count < :velocity`. The whole hot path is one network round trip.
- **`gc-ledger`** — on-demand, PITR on. Partition key `code` (string), sort key `ts#seq` (ISO timestamp plus a sequence suffix) so a card's events sort in order and are unique. Attributes per row: `event` (`issue` / `redeem` / `expire` / `freeze`), `amount` (cents), `balance_after`, and source metadata. Append-only by convention — no function ever updates or deletes a ledger row, and IAM denies `DeleteItem` on this table.

S3, SES, SQS, and Secrets Manager

- **S3** — one versioned bucket holding monthly reports, CSV exports, and periodic ledger snapshots. Versioning means each month's close is preserved even if a later write lands on the same key.
- **SES** — outbound only: issue receipts, redemption receipts (with remaining balance), freeze alerts to the owner, and the monthly report.
- **SQS + DLQ** — buffers receipt and alert sends so a transient SES hiccup never blocks a redemption; anything that fails repeatedly lands in the dead-letter queue for inspection.
- **Secrets Manager** — one secret holding SES and configuration credentials, read at cold start.

IAM, observability, and region

- **IAM** — one execution role per function, scoped to exactly what it touches. `gcr-issue` gets `PutItem` / `TransactWriteItems` on the two tables; `gcr-redeem` gets `UpdateItem` on `gc-cards` and `PutItem` on `gc-ledger`; `gcr-expiry` gets `Query` / `UpdateItem` on `gc-cards` and `PutItem` on `gc-ledger`; `gcr-report` gets read-only `Query` / `Scan` plus S3 `PutObject`, SES `SendEmail`, and Bedrock `InvokeModel`. `DeleteItem` on `gc-ledger` is granted to nobody.
- **CloudWatch Logs** — 7-day retention on every function. **AWS Budgets** — two budgets with an email alert, so a runaway cost is caught early.
- **Region** — everything in `eu-west-2`. No API Gateway, no NAT Gateway, no VPC-bound compute, no always-on anything.

THE WHOLE SYSTEM IN ONE BREATH

- Four Lambdas, two tables, two schedules, one region, one secret.
- Issue is a two-write transaction; redeem is one conditional `UpdateItem` doing the balance and all three caps at once.
- The ledger is append-only and `DeleteItem` is granted to no role — history cannot be rewritten.
- Bedrock appears once a month for prose; every cent is computed in plain Python over integers.
- About $2.10/month at SMB volume, and nothing in the design grows into a server.