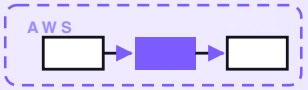
7-PART SERIES · FREE COMPANION



Recurring invoice generator

Every business that runs on contracts and retainers bills the same customers the same amounts on the same dates — and doing it by hand is how an invoice goes out three days late, with last month's figure, missing the mid-cycle change nobody remembered. This is the design of a small serverless generator that turns each contract into a billing schedule, builds the invoice on the day it's due, applies proration and the right tax in plain code, renders a clean PDF, emails it, and records it in a ledger. The arithmetic is deterministic — no model ever touches the money — and an overdue invoice is handed to a separate chaser rather than nagged here. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

BUILD IT FOR REAL

Free lite starter + this PDF · paid tiers at

shop.allanninal.dev/w/recurring-invoice-generator

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

CONTENTS

Recurring invoice generator

PART 1 OF 7

JUNE 26, 2026 PART 1 OF 7 · RECURRING INVOICE GENERATOR SERIES ~10 MIN READ

A recurring invoice generator on AWS for a few dollars a month

A business that runs on retainers bills the same customers, the same amounts, on the same dates, month after month — and that very regularity is what makes it slip. The invoice that goes out three days late because someone was on holiday; the one still on last quarter's figure because the price rose and nobody updated the template; the mid-month upgrade that never made it onto a bill at all. This post walks through the design of a small generator that turns each contract into a schedule and bills it, to the penny, on the day it's due.

KEY TAKEAWAYS

- Each contract or retainer defines its line items, its cycle — monthly, quarterly, or per-milestone — and the customer's jurisdiction.
- Every contract gets its own EventBridge Scheduler entry that fires on the due date and builds that invoice; nothing runs in between.
- Proration, tax, and totals are plain, deterministic code. No model ever touches a number on the invoice.
- A built invoice is rendered to a PDF, emailed through SES, and written to a ledger — every figure auditable.
- Designed on AWS for about $2.00/month at roughly 120 invoices a month. Overdue invoices are handed to a separate chaser, never nagged from here.

The whole system on one page

Before any code, here's the shape of what we're designing. A business that runs on retainers and contracts bills the same customers, the same amounts, on the same dates, month after month. That regularity ought to make it the easiest job in the office — and yet it's the one that slips. The invoice that goes out three days late because the person who sends it was away. The one still showing last quarter's price because the rate rose and the template didn't. The mid-month upgrade that never reached a bill because nobody did the half-month maths. The system below does exactly one thing: it bills each contract, to the penny, on the day it's due — and then gets out of the way.

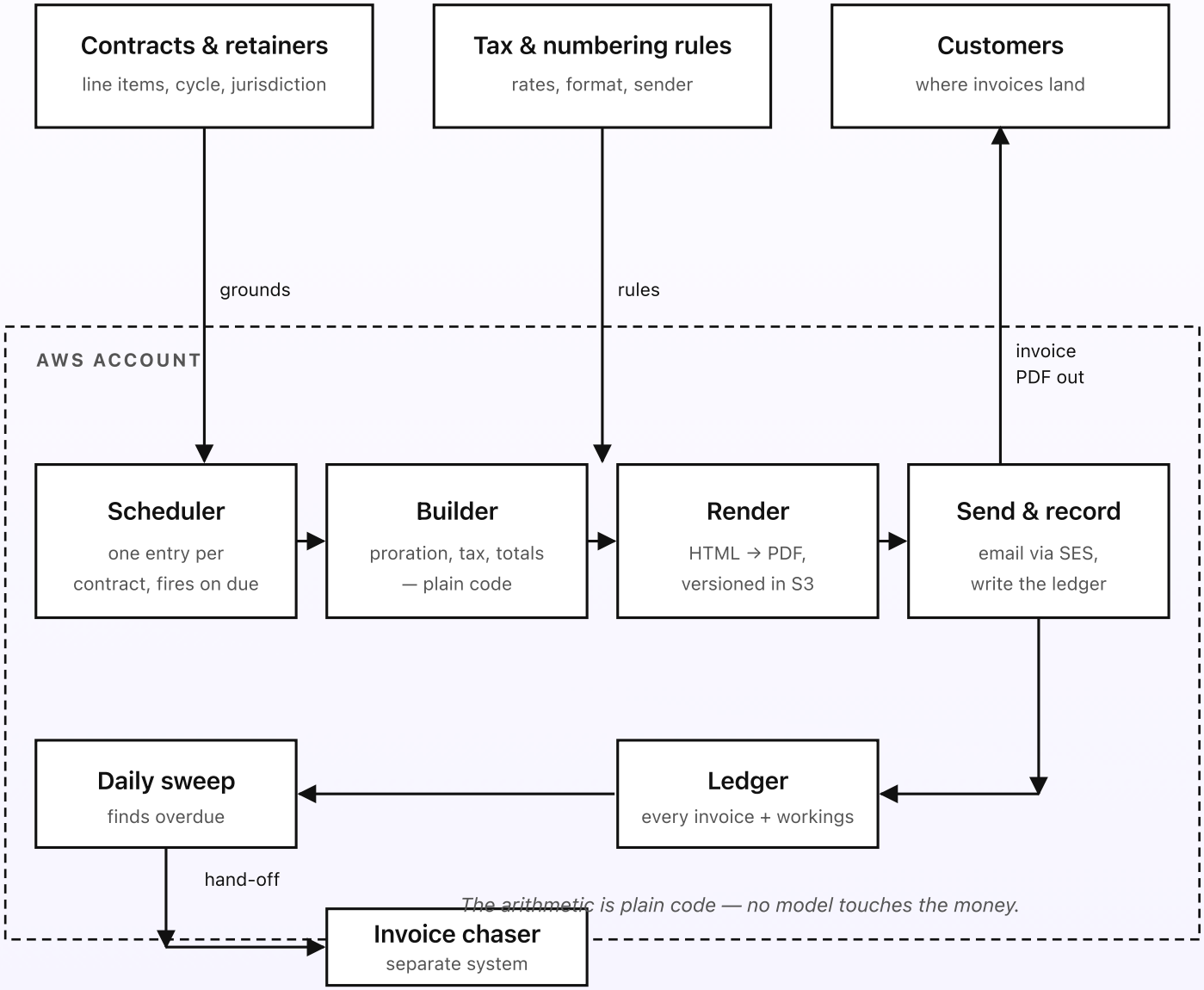

Contracts & retainers
line items, cycle, jurisdiction
Tax & numbering rules
rates, format, sender
Customers
where invoices land
grounds
rules
AWS ACCOUNT
invoice
PDF out
Scheduler
one entry per
contract, fires on due
Builder
proration, tax, totals
— plain code
Render
HTML → PDF,
versioned in S3
Send & record
email via SES,
write the ledger
Daily sweep
finds overdue
Ledger
every invoice + workings
hand-off
The arithmetic is plain code — no model touches the money.
Invoice chaser
separate system

Fig 1. Contracts and rules outside, four pieces inside AWS. A per-contract scheduler fires on each due date; the builder does the arithmetic in plain code; render makes the PDF; send emails it and writes the ledger. Overdue invoices are handed to a separate chaser.

What you set up once (the outside)

- **Contracts and retainers.** A Google Sheet in a Drive folder with one row per contract: the customer, their email, the line items (a description, a quantity, and a unit price for each), the billing cycle (monthly, quarterly, or per-milestone), the anchor date the cycle counts from, the term, and the customer's tax jurisdiction. This is the same list a finance person already keeps in their head or in a spreadsheet; here it's in one place the generator can read. A small sync mirrors it into AWS so the schedules and the builder always work from the current version. How a row becomes a live schedule is Part 2.
- **Tax and numbering rules.** One short settings doc in the same folder. It holds the tax rate per jurisdiction (UK VAT 20%, Ireland 23%, a zero-rated state, and so on), the invoice-numbering format (a prefix, a running sequence, the reset rule), your sender name and address for the PDF, and the payment terms. Changing a VAT rate or the numbering format is an edit to this doc, not a deploy.
- **Customers.** The people who receive the invoices. Each finished invoice goes out as a PDF attached to a plain, templated email from your verified sending address, with a short covering line and a link to view it online. The generator only ever *sends* the invoice — it never takes payment, and it never follows up. That second job belongs to a separate chaser, by design.

What runs each cycle (the inside)

- **The scheduler.** Every contract gets its own EventBridge Scheduler entry, set to fire on its next due date according to its cycle. A monthly retainer anchored to the 1st fires on the 1st; a quarterly contract fires every three months from its anchor; a per-milestone contract fires when its milestone is marked done. Nothing polls and nothing runs between cycles — the schedule simply wakes the builder on the right day. This is Part 2.
- **The builder.** When a schedule fires, the builder loads the contract, pulls its line items, applies proration for any change that happened part-way through the cycle, looks up the tax rate for the customer's jurisdiction, computes every line total and the invoice total, and assigns the next number in sequence. All of this is plain Python — the same inputs always produce the same invoice. This is Parts 3 and 4.
- **Render and send.** The built invoice is poured into an HTML template and rendered to a PDF inside Lambda, then written to a versioned S3 bucket so every issued document is kept exactly as it was sent. SES emails it to the customer, and the invoice — with its full workings — is written to the ledger in DynamoDB. This is Part 5.
- **The sweep and the hand-off.** A daily sweep re-checks invoices that are due and still unpaid. It doesn't nag anyone. It marks each overdue invoice in the ledger and pushes it onto a queue for a separate invoice chaser, with the invoice id, customer, amount, and due date attached. Keeping generation and chasing apart keeps this system a clean, predictable biller. Also in Part 5.

In plain words

It's the 1st of July. A design studio bills a client on a £1,500/month retainer, anchored to the 1st. On 16 June that client added a second service line worth £600/month. At one minute past midnight the contract's schedule fires. The builder loads the contract, sees the base retainer (£1,500 for July), the new line (£600 for July), and a proration catch-up for the half-month the new line ran in June — £600 × 15/30 = £300. That's a net subtotal of £2,400. The client is in the UK, so VAT at 20% adds £480, for a total of £2,880. The builder assigns invoice `INV-2026-0148`, renders the PDF, and SES emails it before anyone is awake. The ledger now holds the invoice, the line-by-line workings, and the exact proration sum — so when the client queries it, the answer is right there.

Three weeks later that invoice is still unpaid. The generator does nothing dramatic. The daily sweep notices it has passed its due date, marks it *overdue* in the ledger, and drops a small message onto the chaser's queue. The chaser — a separate system with its own tone and escalation rules — takes it from there. The generator is already on to next month.

DESIGN RULES THAT SHAPED EVERY DECISION

- One job. The generator builds and sends scheduled invoices — it does not chase, quote, or take payment.
- No model touches the money. Every figure is plain, deterministic code that produces the same invoice every time.
- One schedule per contract. Each contract's cycle is its own EventBridge entry; nothing polls and nothing runs between cycles.
- Every invoice is kept exactly as sent. The PDF is versioned in S3 and the workings are written to the ledger.
- Rules live in a doc. Tax rates, the numbering format, and the sender details change without a deploy.
- Overdue is someone else's job. Anything unpaid is handed cleanly to a separate chaser, never nagged from here.

Why this shape

Most small teams run recurring billing one of three ways: a person re-uses last month's invoice and edits the date, the accounting package fires a fixed template that can't handle a mid-cycle change, or it's all in someone's head and goes out when they remember. Editing last month's copy is how a stale price survives for a year. A rigid template is how a half-month upgrade gets billed as nothing or as a whole month. And "when someone remembers" is how cash flow develops a stutter. None of these keep a clean record of *why* a number is what it is — which is exactly what you want the day a client disputes a line.

The shape above keeps the contract you already have as the single source of truth, turns it into a schedule that fires on the right day on its own, and does the arithmetic the same boring way every time — in code you can read, with the workings saved. The 95% of invoices that are simply “this month, same as last” go out untouched and on time. The few with a mid-cycle change are prorated correctly to the penny. And because the generator refuses to also be a chaser, it stays small enough to trust.

The next four posts walk through each piece in turn: how a billing schedule gets set up, how an invoice gets built, how proration and tax get applied (with the numbers), and how an invoice gets sent and handed off. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 26, 2026 PART 2 OF 7 · RECURRING INVOICE GENERATOR SERIES ~7 MIN READ

How a billing schedule gets set up

Before a single invoice can go out on time, the system has to know exactly when each one is due — and “monthly” hides a surprising amount of detail. This post is about that step alone: how a contract’s cycle, anchor date, and term become a concrete schedule, and how each contract gets its own EventBridge Scheduler entry that fires on the right day and nothing more.

KEY TAKEAWAYS

- A contract’s cycle, anchor date, and term are turned into a concrete next-due date — “monthly” on its own isn’t enough.
- Each contract gets its own EventBridge Scheduler entry, set to fire once on its next due date, then rescheduled for the one after.
- Three cycle types are supported: monthly, quarterly, and per-milestone — the first two by date, the third by an event.
- A signup part-way through a cycle is prorated on the first invoice; the schedule itself stays anchored to the clean date.
- A daily catch-up sweep is the safety net — if a fire is ever missed, the next sweep builds the invoice that should have gone out.

From a contract to a due date

“Bill them monthly” sounds like a complete instruction until you try to turn it into a date. Monthly from when — the day they signed, or the 1st? Does a contract signed on the 31st bill on the 31st of a month that only has 30 days? Does a quarterly contract anchored to 15 January fire in April, or three calendar months later to the day? The schedule is the part of the system that has to answer all of this concretely, because a scheduler doesn’t take “monthly” — it takes a timestamp.

Each contract carries three fields that pin its timing down. The *cycle* says how often: monthly, quarterly, or per-milestone. The *anchor date* says what the cycle counts from — usually the day the contract starts, but it can be normalised to a clean day like the 1st so a whole book of contracts bills together. The *term* says when to stop — an end date, a fixed number of cycles, or open-ended until cancelled. From those three, a small piece of plain code computes one thing: the next date this contract is due. That date, and only that date, is what goes to the scheduler.

The three cycle types

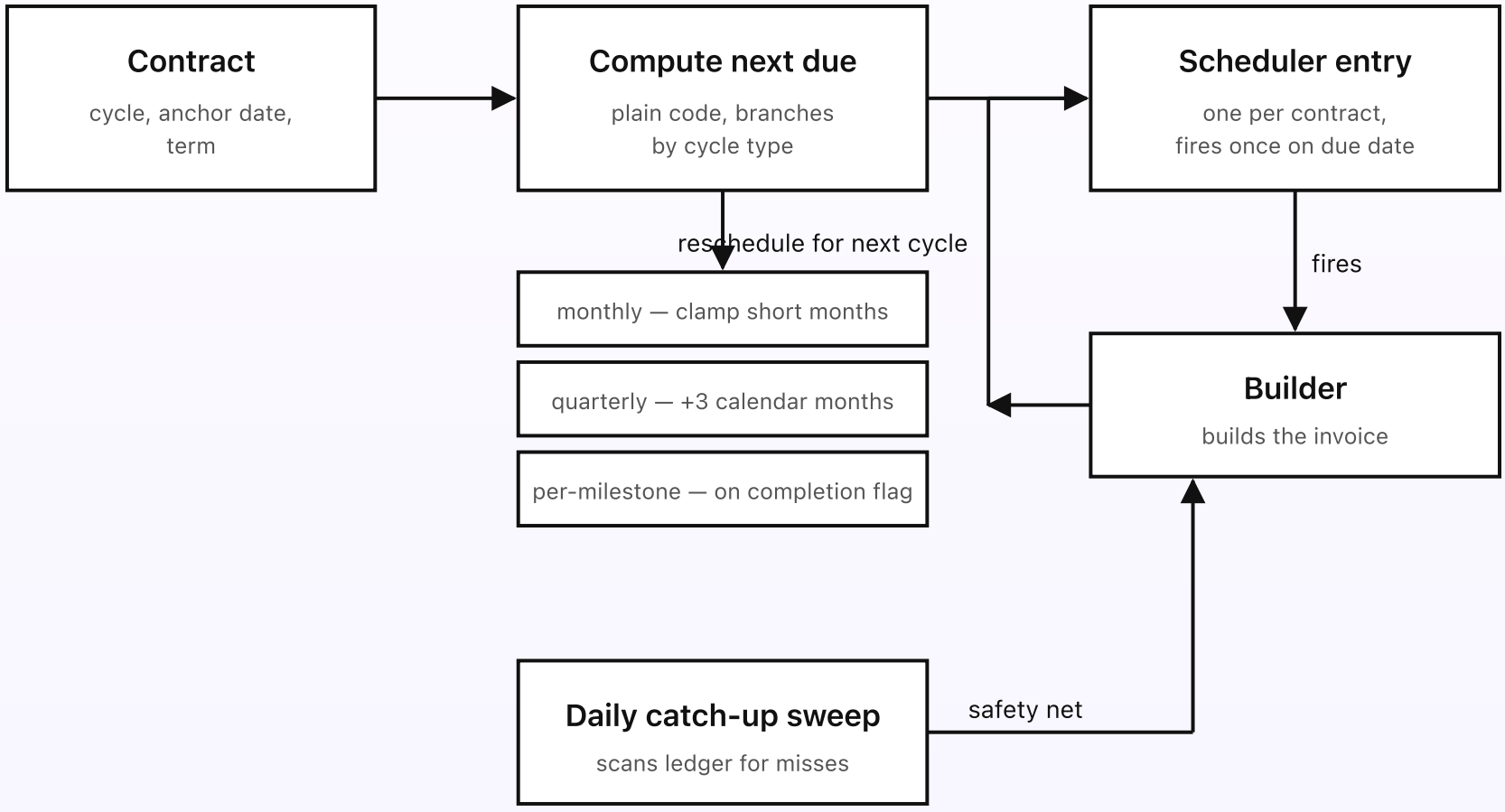
- **Monthly.** The common retainer. Anchored to a day of the month; if that day doesn’t exist in a shorter month (the 31st in February), it clamps to the last day. A retainer anchored to the 1st bills on the 1st of every month; one anchored to the signup day of the 18th bills on the 18th.
- **Quarterly.** Every three months from the anchor, by calendar month rather than by 90 days, so a contract anchored to 15 January bills on 15 January, 15 April, 15 July, and 15 October — predictable dates a customer can plan around.

- **Per-milestone.** No calendar at all. The contract lists milestones (a phase, a deliverable, a stage payment), and an invoice is due when a milestone is marked complete — a flag flipped in the contract sheet. This is the project-work case, where billing follows delivery rather than the calendar.

One schedule per contract

The obvious design is a single nightly job that scans every contract and asks “is anything due today?” It works, but it scans the whole book every night to act on a handful of rows, and a bug in the scan can miss or double-bill across all of them at once. This system does the opposite: each contract gets its own EventBridge Scheduler entry, set to fire *once*, at the exact next due date. When that fire builds the invoice, the last step is to compute the following due date and reschedule the same entry for it. The schedule walks itself forward, one cycle at a time, and a contract that isn’t due simply has no activity that day.

Per-milestone contracts don’t get a date-based entry; they get a small trigger that fires when the milestone flag flips during a sync. Either way, the unit of scheduling is the contract, so pausing, cancelling, or changing one contract’s timing never touches another’s.



The schedule walks itself forward one cycle at a time; the daily sweep catches anything missed.

Fig 2. A contract's cycle, anchor, and term become one next-due date. EventBridge fires the builder on that date, which reschedules the entry for the following cycle. A daily sweep is the safety net for any missed fire.

Signing up mid-cycle

A contract rarely starts on a tidy date. A client signs a £1,200/month retainer on 18 June and wants to be billed on the 1st like everyone else. There are two clean ways to handle the gap, and the contract's setup chooses one. Either the anchor is the signup day (the 18th) and they simply bill on the 18th forever — no proration needed. Or the anchor is normalised to the 1st and the first invoice is *prorated*: June has 30 days, the client was active from the 18th to the 30th, that's 13 days, so the first bill is £1,200 × 13/30 = £520, and every invoice after that is the clean £1,200 on the 1st. Either way the *schedule* stays simple — it always points at a clean recurring date. Proration is a builder concern, not a scheduler one, and it's exactly what Part 4 works through with numbers.

Why a daily sweep as well

Per-contract schedules are precise, but "fire exactly once on the right day" is a promise worth backing up. Things go wrong: a deploy lands mid-fire, a downstream queue is briefly unavailable, a schedule gets left paused after maintenance. So one more EventBridge entry — a single daily catch-up sweep — scans the ledger each morning and asks a blunt question: is there any contract whose due date has passed without an invoice in the ledger? If so, it builds the one that's missing. Because every invoice carries the contract id and the cycle it belongs to, a re-run can never produce a duplicate — the builder checks the ledger before it writes, so the sweep can only ever fill a genuine gap. It's the

cheapest possible insurance: one extra scheduled function whose whole job is to make sure nothing silently fails to bill.

WHY THIS SHAPE

- The scheduler only ever holds dates. “Monthly” is resolved to a concrete next-due date in plain code before anything is scheduled.
- One entry per contract. Pausing or changing one contract’s timing never disturbs another’s.
- The schedule self-advances. Each fire reschedules itself for the following cycle, so the book stays current with no central scan.
- Milestones are events, not dates. Per-milestone billing fires on a completion flag, keeping project work out of the calendar logic.
- A daily sweep is the safety net. If a fire is ever missed, the sweep builds the missing invoice — and the ledger check makes a duplicate impossible.

PART 3 OF 7

JUNE 26, 2026 PART 3 OF 7 · RECURRING INVOICE GENERATOR SERIES ~7 MIN READ

How an invoice gets built

When a schedule fires, the contract is just a set of line items and a customer. This post is about the step that turns that into an actual invoice: where the line items come from, how the invoice number is assigned without ever colliding or skipping, and why every figure on the document is computed by plain code that produces the same answer every single time.

KEY TAKEAWAYS

- When a schedule fires, the builder loads the contract, gathers its line items, and assembles one complete invoice in plain code.
- The invoice number comes from an atomic counter in DynamoDB — it never collides and never skips, even on a retry.
- The builder is idempotent: building the same contract-and-cycle twice produces the same invoice, never a duplicate in the ledger.
- Bedrock is optional and only ever polishes a line's description — it never sees or sets a quantity, a price, or a total.
- The finished invoice and its full workings are written to the ledger before anything is rendered or sent.

What the builder receives

A schedule firing hands the builder almost nothing — a contract id and the cycle date it's billing for. That deliberate thinness is what makes the build trustworthy: everything the invoice contains is read fresh from the current contract, not carried over from last month's document. The builder loads the contract from the mirrored store, reads its line items, its customer, and its jurisdiction, and from there does the same handful of steps every time — gather, number, compute, record — with no branch that depends on anything outside the contract and the rules doc.

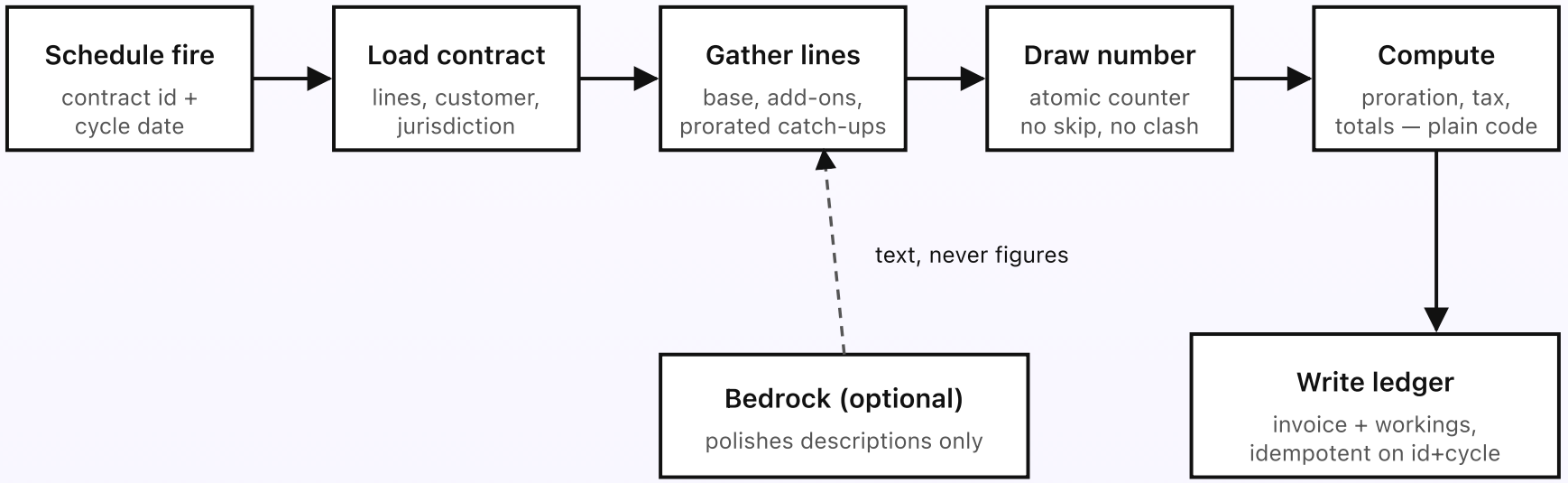
The first step is gathering the lines. A retainer is often a single line ("Monthly retainer", quantity 1, the agreed price). Most contracts have a few: a base retainer, a couple of add-on services, perhaps a usage line the contract pulls from a counter. Each line has a description, a quantity, and a unit price. The builder also checks whether anything changed part-way through this cycle — a line added, removed, or repriced since the last invoice — because those are the lines that need proration. The arithmetic of that is Part 4; here the point is simply that the builder collects every line that belongs on this invoice, including the prorated catch-up lines, before it adds anything up.

The invoice number

An invoice number has two hard requirements that fight each other: it must be unique, and for most tax regimes it must be sequential with no gaps. A naive "read the highest number, add one, write it back" breaks the moment two builds run close together or a retry re-enters — you get a collision or a skip. So the number

comes from a single counter item in DynamoDB, incremented with an atomic update that returns the new value in one operation. Two builds can never be handed the same number, and the sequence never jumps. The format itself — the prefix, the width, whether it resets each year — comes from the numbering rule in the settings doc, so `INV-2026-0148` is just the counter dressed in the format you chose.

The number is only stamped onto an invoice that is actually going to be issued. If a build is abandoned before it commits — a contract turns out to be paused, say — no number is drawn, so the sequence doesn't develop a hole for an invoice that never existed.



Same inputs, same invoice, every time — the build is idempotent and no model touches a number.

Fig 3. The builder's path: load the contract, gather every line, draw an atomic invoice number, compute the figures in plain code, and write the invoice and its workings to the ledger. Bedrock, if used, touches descriptions only.

Idempotent by design

Two things can ask the builder to build the same invoice: a retry after a transient failure, and the daily catch-up sweep from Part 2. Both are good behaviours — you *want* the system to try again rather than silently drop a bill — but they mean the builder has to be safe to run twice. It is, because the ledger record is keyed by the contract id and the cycle it bills. Before the builder commits a new invoice it does a conditional write on that key: if an invoice already exists for this contract and this cycle, the write is rejected and the builder simply returns the one that's already there. A retry can never produce two invoices for the same month, and the sweep can only ever fill a genuine gap. The single risk in recurring billing — billing the same period twice — is closed at the database, not in fragile application logic.

Where the model is allowed, and where it isn't

This system can use Bedrock, but only in one narrow, optional place: turning a terse contract line into a clearer description. A contract might store a line as "Apr retainer — phase 2", and a contract that opts in can have the model expand that into "Monthly retainer, April 2026 — Phase 2 delivery and support" for the printed invoice. That is the entire remit. The model never sees a price, never sets a quantity, never picks a tax rate, and never computes a total. Every figure on the invoice is plain Python operating on the contract's own numbers. Switch Bedrock off entirely and the invoice still renders correctly — the descriptions are just the raw contract text. Keeping the model strictly on the wording side of the line is what lets the arithmetic be audited and trusted; a number you can't reproduce by hand has no place on an invoice.

By the time the builder finishes, the ledger holds a complete, numbered invoice with every line, the tax, the total, and the exact workings behind each figure. Nothing has been rendered or sent yet — recording comes first, so an invoice can never go out without a matching record. Turning that record into a PDF in someone's inbox is Part 5; the proration and tax arithmetic it depends on is Part 4.

WHY THIS SHAPE

- Build from the contract, not last month. Everything on the invoice is read fresh, so a stale price can't survive.
- Numbers come from an atomic counter. Unique and gap-free, even under retries and concurrent builds.
- Record before you render. The ledger write commits first, so no invoice can be sent without a matching record.
- Idempotent on contract-and-cycle. Building the same period twice returns the same invoice — never a duplicate.
- The model stays on the wording. Bedrock is optional and touches descriptions only; every figure is plain, reproducible code.

PART 4 OF 7

JUNE 26, 2026 PART 4 OF 7 · RECURRING INVOICE GENERATOR SERIES ~7 MIN READ

How proration and tax get applied

Proration and tax are where invoices quietly go wrong — a half-month billed as a whole one, the wrong VAT rate for the customer's country, a rounding penny that won't reconcile. This post does one thing: walks a real invoice through proration and tax line by line, with the actual figures, so you can see exactly how the deterministic arithmetic lands — and why none of it is left to a model.

KEY TAKEAWAYS

- Proration is a calendar-day calculation: a line is billed for the days it was active over the days in the cycle.
- Tax is a lookup against the contract's jurisdiction, applied to the net subtotal — UK 20%, Ireland 23%, a zero-rated state, and so on.
- The rounding rule is stated once and applied everywhere: each line nets to the penny, tax rounds half-up on the subtotal.
- Every figure is plain Python operating on the contract's own numbers — reproducible by hand, never set by a model.
- The full workings — the proration fraction, the rate used, every rounded figure — are written to the ledger alongside the invoice.

A real invoice, end to end

Proration and tax are where invoices quietly go wrong — a half-month billed as a whole one, the wrong VAT rate for the customer's country, a stray penny that won't reconcile against the ledger. The way to trust the arithmetic is to watch it run on a real case, so this whole post is one worked example, with the actual numbers.

The contract: a design studio bills a UK client a **£1,500/month** retainer, anchored to the 1st. On **16 June** the client adds a second service line worth **£600/month**. It's now 1 July, and the contract's schedule has just fired the builder to produce the July invoice. The interesting part is that this invoice has to settle the half-

month of June that the new line ran before the next clean cycle, as well as bill July normally.

Step one: proration for the mid-cycle change

The new £600 line started on 16 June, part-way through a cycle that had already been billed at £1,500 on 1 June. So that line owes a catch-up for the days it was active in June. Proration is a plain calendar-day fraction:

```
days active ÷ days in cycle × line price
```

June has **30** days. The line was active from the 16th to the 30th inclusive — **15** days. So the June catch-up is **£600 × 15/30 = £300.00**. The fraction is computed from real calendar days, not an assumed 30, so a line that starts mid-February or mid-July prorates against the correct month length. (Some businesses prefer a fixed 30-day month for retainers; that's a one-line rule in the settings doc — the point is that it's a stated, deterministic choice, not a guess.)

Step two: the lines on the July invoice

The builder gathers three lines for this invoice — the two full-month July lines and the one June catch-up:

Line	Working	Net
Monthly retainer — July	£1,500 × 1	£1,500.00
Additional service — July	£600 × 1	£600.00
Additional service — June (16–30, prorated)	£600 × 15/30	£300.00

Line	Working	Net
Net subtotal		**£2,400.00**

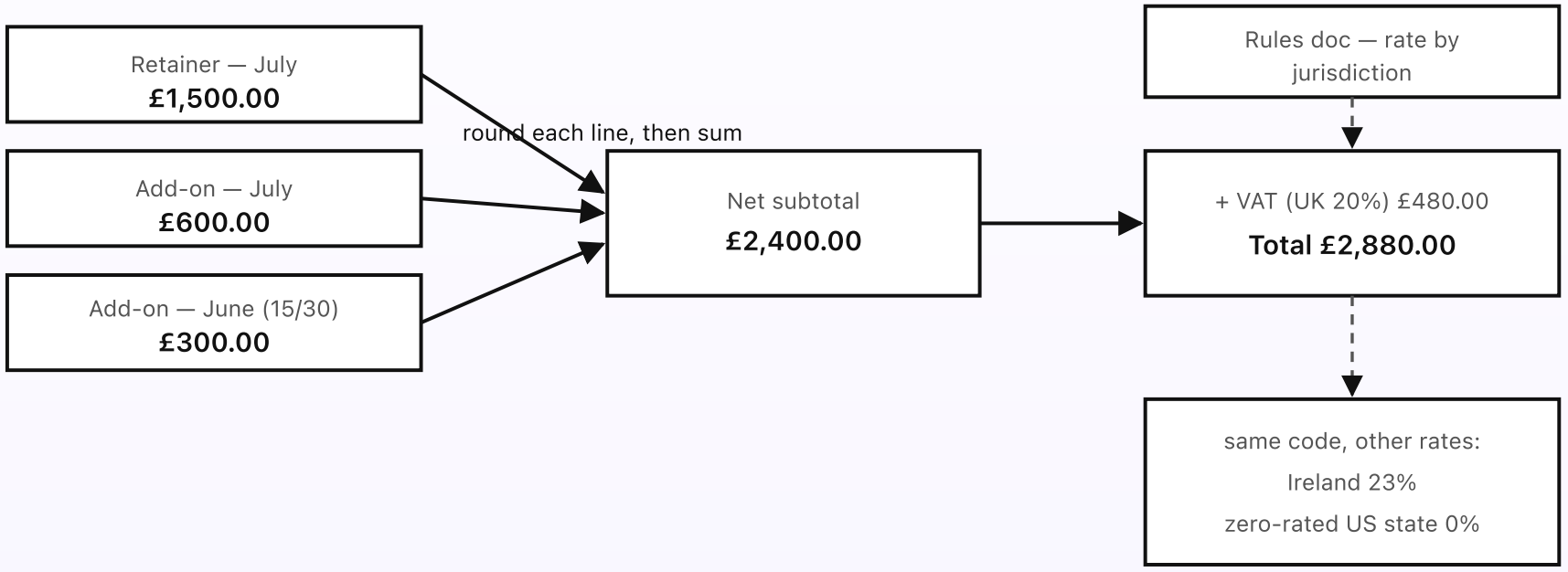
Each line is rounded to the penny first, then the lines are summed to the net subtotal. Rounding per line and then summing — rather than carrying fractions of a penny into the total — is the rule that keeps the printed lines adding up to the printed subtotal, which is the first thing anyone checks.

Step three: tax by jurisdiction

The customer's jurisdiction on the contract is the United Kingdom, so the builder looks up the UK rate in the settings doc — VAT at **20%** — and applies it to the net subtotal:

Component	Working	Amount
Net subtotal		£2,400.00
VAT (UK, 20%)	£2,400.00 × 0.20	£480.00
Invoice total		**£2,880.00**

Tax is applied to the net subtotal, rounded half-up to the nearest penny. The rate is never hard-coded — it's read from the rules doc against the jurisdiction, so the same builder produces the right figure for a client anywhere you do business.



Every figure is plain code on the contract's own numbers; the full workings go to the ledger.

Fig 4. The July invoice end to end: three lines (two full, one prorated 15/30) sum to a £2,400.00 net subtotal; UK VAT at 20% adds £480.00 for a £2,880.00 total. Only the looked-up rate changes for another jurisdiction.

The same code, other jurisdictions

Nothing about the arithmetic changes when the customer is somewhere else — only the rate the builder looks up. The identical net subtotal of £2,400.00 produces £552.00 of tax for an Irish client (VAT 23%, total £2,952.00), and £0.00 of tax for a client in a US state that doesn't tax this service (total £2,400.00). Because the rate lives in the rules doc keyed by jurisdiction, a rate change — or a new jurisdiction — is a doc edit, not a deploy, and it can never be quietly wrong in a way that only some invoices inherit. The one rule worth being strict about is that tax is computed from the jurisdiction on the contract, never inferred — if a contract's jurisdiction is missing or unknown, the builder stops and flags it rather than guessing a rate.

Why none of this is a model's job

It would be easy to hand a language model the contract and ask it to "work out the invoice", and it would usually be right. "Usually right" is exactly the problem with money. Proration is a fraction; tax is a multiplication; rounding is a rule you state once. All three are reproducible by anyone with a calculator, which means an invoice can be defended, reconciled, and audited — and a wrong figure can be traced to a wrong input, not to a model that had an off day. So the builder writes every step into the ledger alongside the invoice: the proration fraction (15/30), the rate used (UK 20%), and each rounded figure. When the client asks why July was £2,880 and not the usual £1,800, the answer isn't "the system decided" — it's a three-line explanation anyone can check. That is the whole reason no model goes near the numbers.

WHY THIS SHAPE

- Proration is calendar days. Active days over days in the cycle, against the real month length unless the rules doc says otherwise.
- Tax is a lookup, not a guess. The rate comes from the contract's jurisdiction; a missing jurisdiction stops the build.
- Round once, the same way. Each line nets to the penny, then tax rounds half-up on the subtotal — so the printed figures add up.
- The rate lives in a doc. New jurisdictions and rate changes are edits, never deploys.
- The workings are saved. The fraction, the rate, and every rounded figure go to the ledger, so any invoice can be explained.

PART 5 OF 7

JUNE 26, 2026 PART 5 OF 7 · RECURRING INVOICE GENERATOR SERIES ~7 MIN READ

How an invoice gets sent and handed off

A finished invoice still has to become a real PDF in someone's inbox, and then it has to be tracked. This post is about the last stretch: rendering the document, sending it through SES, recording it in the ledger as sent, and — the part most generators get wrong — doing nothing further when it goes overdue, except hand it cleanly to a separate chaser that owns the follow-up.

KEY TAKEAWAYS

- The recorded invoice is poured into an HTML template and rendered to a PDF inside Lambda — no external rendering service.
- Every rendered PDF is written to a versioned S3 bucket, so the exact document sent to a customer is kept forever.
- SES emails the PDF to the customer with a short covering note and a view link; the ledger is updated to *sent* only after SES accepts it.
- A daily sweep marks any invoice past its due date as *overdue* — it does not nag, email, or escalate.
- Overdue invoices are handed to a separate chaser through an SQS queue, with the id, customer, amount, and due date attached.

From a record to a PDF

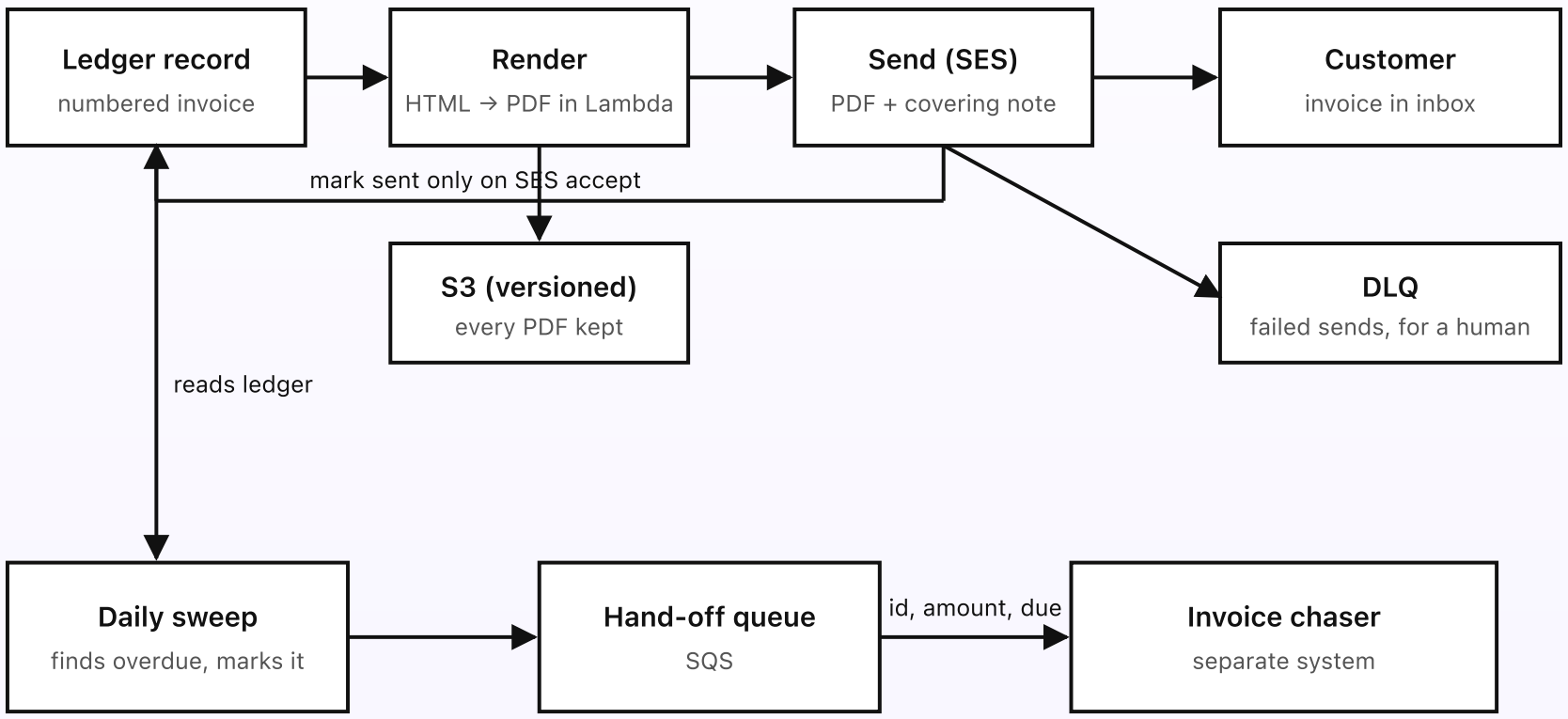
By the time this stage runs, the ledger already holds a complete, numbered invoice with every line and its workings (Parts 3 and 4). What it doesn't have yet is a document a customer can actually open. Rendering is deliberately the dullest step in the system: the invoice record is poured into a fixed HTML template — your letterhead, the customer's details, the line table, the net subtotal, the tax line, the total, the payment terms — and that HTML is converted to a PDF inside the Lambda itself, with a small HTML-to-PDF library packaged in a layer. There's no headless browser farm and no third-party rendering API to depend on or pay for; the whole conversion happens in the same function, in a fraction of a second, and the only input is the ledger record, so the PDF is a faithful printout of figures that were already computed and saved.

The finished PDF is written to a versioned S3 bucket under a key derived from the invoice number. Versioning matters here for a specific reason: an invoice is a legal-ish document, and the copy the customer received should be retrievable exactly as it went out, even if you later reissue or correct it. With versioning on, a reissue writes a new version rather than overwriting the original, so the audit trail — what was sent, when — is intact by default rather than by discipline.

Sending it

SES emails the PDF to the customer from your verified sending domain, with a short, templated covering message — the invoice number, the amount, the due date, and a link to view it online — and the PDF attached. The order of operations is the careful part. The ledger record is only moved to *sent*, with the timestamp

and the SES message id, *after* SES has accepted the message for delivery. If SES fails, the record stays in its pre-send state and the work goes back onto the queue to retry; after a few attempts it lands in the dead-letter queue for a human to look at, rather than being silently lost. The result is that the ledger's "sent" flag means what it says — an invoice marked sent really did leave the building — which is precisely the flag the overdue logic will rely on.



The generator sends and records, then stops — chasing belongs to a separate system.

Fig 5. Render the PDF, store it versioned, send by SES, and mark the invoice sent only once SES accepts it. The daily sweep marks overdue invoices and hands them to a separate chaser through a queue — it never nags from here.

Overdue is a clean hand-off, not a nag

Here is the decision that keeps this system small. When an invoice passes its due date unpaid, the generator does almost nothing. A daily EventBridge sweep reads the ledger, finds every invoice that is marked *sent*, is past its due date, and isn't marked paid, and changes its status to *overdue*. Then — and this is the whole point — it stops. It does not send a reminder, it does not email the customer, it does not start a polite-then-firm sequence. It simply pushes a small message onto an SQS hand-off queue: the invoice id, the customer, the amount, and the due date. A separate invoice chaser, with its own tone, its own escalation ladder, and its own record of who's been contacted, picks it up from there.

Splitting the two is a deliberate boundary. Generating an invoice and chasing one are different jobs with different rhythms: generation is a clean, once-a-cycle event that must be exactly right; chasing is a judgement-heavy, multi-step conversation that depends on relationships and history. Bolting the chaser onto the generator would make the generator harder to reason about and would couple a system you want to be boringly predictable to one that's necessarily fiddly. By handing overdue invoices across a queue, the generator stays a clean biller, the chaser owns all the follow-up, and either can be changed without touching the other. (How the chaser actually pursues a payment — the cadence, the wording, the escalation — is its own system, and its own series.)

What "paid" looks like

The generator doesn't take payment, so it learns an invoice is settled the same way it learns anything else — from the contract source. When a payment is recorded against an invoice (in the accounting system or the sheet that feeds the contract store), the next sync marks that ledger row *paid*, and the daily sweep simply stops considering it. There's no race and no special case: an invoice that's already paid before its due date is never marked overdue and never handed to the chaser, because the sweep only ever acts on what the ledger says right now. The generator's entire relationship with payment is to read whether it happened — never to chase it, and never to process it.

WHY THIS SHAPE

- Render in Lambda, store versioned. No external rendering service, and the exact PDF sent is kept forever in S3.
- Mark sent only after SES accepts. The "sent" flag is trustworthy, because nothing else is allowed to set it.
- Failed sends retry, then surface. A dead-letter queue means a send can't silently vanish.
- Overdue is a status change and a hand-off. The generator marks it and queues it — it never nags.
- Chasing is a separate system. Generation stays predictable; follow-up logic lives where it belongs.

PART 6 OF 7

JUNE 26, 2026 PART 6 OF 7 · RECURRING INVOICE GENERATOR SERIES ~6 MIN READ

What the recurring invoice generator costs

A billing system that costs more than the time it saves is a bad joke. This post is the cost breakdown: every AWS service this generator touches, what each one adds up to at around 120 invoices a month, why the total lands near $2.00 — and why the one line worth naming is the one that bills while the system sleeps.

KEY TAKEAWAYS

- About $2.00/month at roughly 120 invoices, and the fixed cost is essentially zero — nothing runs between billing cycles.
- The single largest line is Secrets Manager, at $0.40 per secret per month for the Drive and signing keys.
- The work that builds, prorates, taxes, and renders each invoice is plain Python on Lambda and rounds to cents.
- Bedrock is optional and small — it only polishes some descriptions, so the model line is a fraction of the bill, or zero if switched off.
- At ten times the volume (around 1,200 invoices) the bill lands near $7 — it scales with use, not with idle time.

Where the money goes

The generator is serverless end to end, so there's no instance ticking over between billing runs and no idle bill. You pay for an invoice only when a schedule fires. At a typical small-business volume — call it 120 invoices a month across the contracts you bill — here's the whole bill, line by line.

AWS service	What it does here	Monthly
Secrets Manager	Two secrets — Drive service-account key, view-link signing key ($0.40 each)	$0.80

AWS service	What it does here	Monthly
S3 (versioned)	Every rendered invoice PDF, kept as sent	$0.25
DynamoDB (on-demand)	Contracts mirror, invoice ledger, number counter — small reads and writes	$0.20
CloudWatch Logs	Function logs, 7-day retention	$0.20
Bedrock (Claude Haiku 4.5)	Optional description polish on a fraction of lines	$0.15
Lambda (Python 3.14, arm64)	Builder, render, send, sweep, drive-sync	$0.15
SES	Emailing each invoice PDF to the customer	$0.15
EventBridge Scheduler	Per-contract billing cycles and the daily sweep	$0.05
SQS + DLQ	Buffering between stages, and the overdue hand-off queue	$0.05
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~120 invoices/month**	**$2.00**

The shape of that bill is the point. The single biggest line isn't any of the work — it's Secrets Manager, a flat fee for holding two keys safely, which you'd pay

whether you issued one invoice or a thousand. Everything that actually produces an invoice — building it, prorating it, applying tax, rendering the PDF — is plain Python on Lambda and a few small reads and writes, and it all rounds to cents. Even Bedrock, the one place a model could run, is optional and tiny here, because it only ever tidies up a description on the contracts that ask for it.

The one real fixed cost

It's worth dwelling on Secrets Manager, because it's the only thing here that costs money while the system sleeps. Two secrets at $0.40 each is $0.80 a month no matter what — nearly half the bill at this volume — and it buys you the Drive service-account key (so the contract sync can read your sheet) and the signing key (so each invoice's view link can't be forged or guessed). If you didn't need the view links you'd shave $0.40 off; everything else on the list is genuinely usage-priced and rounds to zero at idle, which is exactly what you want from a system that only does work when an invoice is due.

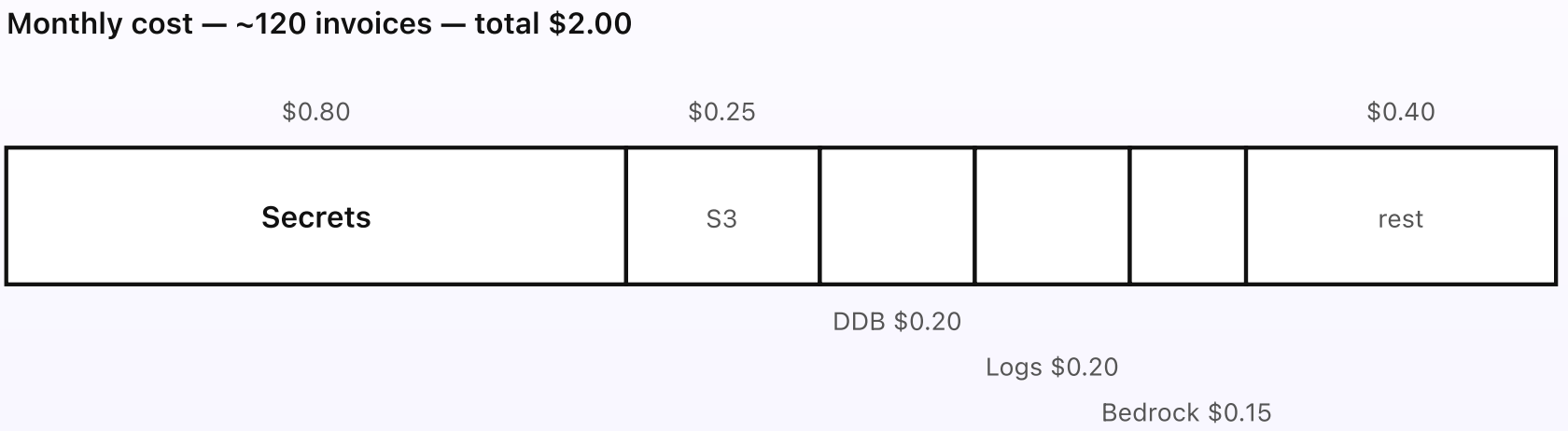


Fig 6. The monthly bill at about 120 invoices. Secrets Manager alone is nearly half of it; everything that builds, taxes, renders, and sends an invoice rounds to cents.

What ten times the volume costs

Push this to a busier book — 1,200 invoices a month, ten times the volume — and the bill lands near $7, not $20. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.80, the schedules barely change (a per-contract entry firing monthly is a rounding error either way), and AWS Budgets stays free.

What scales is the genuinely usage-priced work — more Lambda invocations, more SES sends, more PDFs in S3, a little more Bedrock if you use it — and even at ten times the invoices, the per-invoice work is so cheap that the total is still dominated by that flat secrets fee. Turn Bedrock off and you'd save its line entirely; the invoices would just carry the contract's raw descriptions.

The honest way to read this: the AWS bill is rounding error against the alternative. A person preparing 120 invoices a month by hand — checking each contract, doing the proration, getting the VAT right, rendering and sending — is a steady chunk of someone's week, and the work is exactly the kind that's easy to get subtly wrong when it's dull. $2.00, or even $7, buys that time back and removes the late invoice, the stale price, and the un-prorated upgrade as failure modes — while keeping a perfect record of every figure.

DESIGN RULES THAT SHAPED THE COST

- Pay per invoice, not per hour. No always-on compute means no idle bill between cycles.
- Keep the work in plain code. Building, prorating, taxing, and rendering are Lambda and a few small reads — cents.
- Render in-process. An HTML-to-PDF step inside Lambda avoids a paid rendering service entirely.
- Know your one fixed cost. Secrets Manager is the only line that bills while the system sleeps.
- The model is optional. Bedrock only polishes wording; switch it off and the line disappears.

PART 7 OF 7

JUNE 26, 2026 PART 7 OF 7 · RECURRING INVOICE GENERATOR SERIES ~8 MIN READ

Engineering reference: the recurring invoice generator architecture

This is the recurring invoice generator with the friendly labels removed: the real resource names, the runtime, the scheduler design that gives each contract its own cycle, the table key schemas, the PDF bucket, and the IAM scope. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- Six Lambda functions, all Python 3.14 on arm64, wired through one SQS work queue with a dead-letter queue.
- Three DynamoDB tables, all on-demand: a contracts mirror, an append-only invoice ledger, and a single atomic number counter.
- EventBridge Scheduler holds one entry per contract for its billing cycle, plus one daily catch-up-and-overdue sweep.
- Rendered invoice PDFs live in a versioned S3 bucket; SES sends them; an SQS hand-off queue passes overdue invoices to a separate chaser.
- Bedrock (Claude Haiku 4.5, Global inference) is optional and called only by the builder, for description text. Single region, `eu-west-2`.

The architecture, for engineers

This is the same system as Part 1 with the friendly labels removed and the real resources named. Everything is in one region, `eu-west-2` (London), in one account. There is no API Gateway, no NAT Gateway, and nothing always-on; billing is driven entirely by EventBridge Scheduler, work is buffered on a single SQS queue, and the only outbound surface is SES.

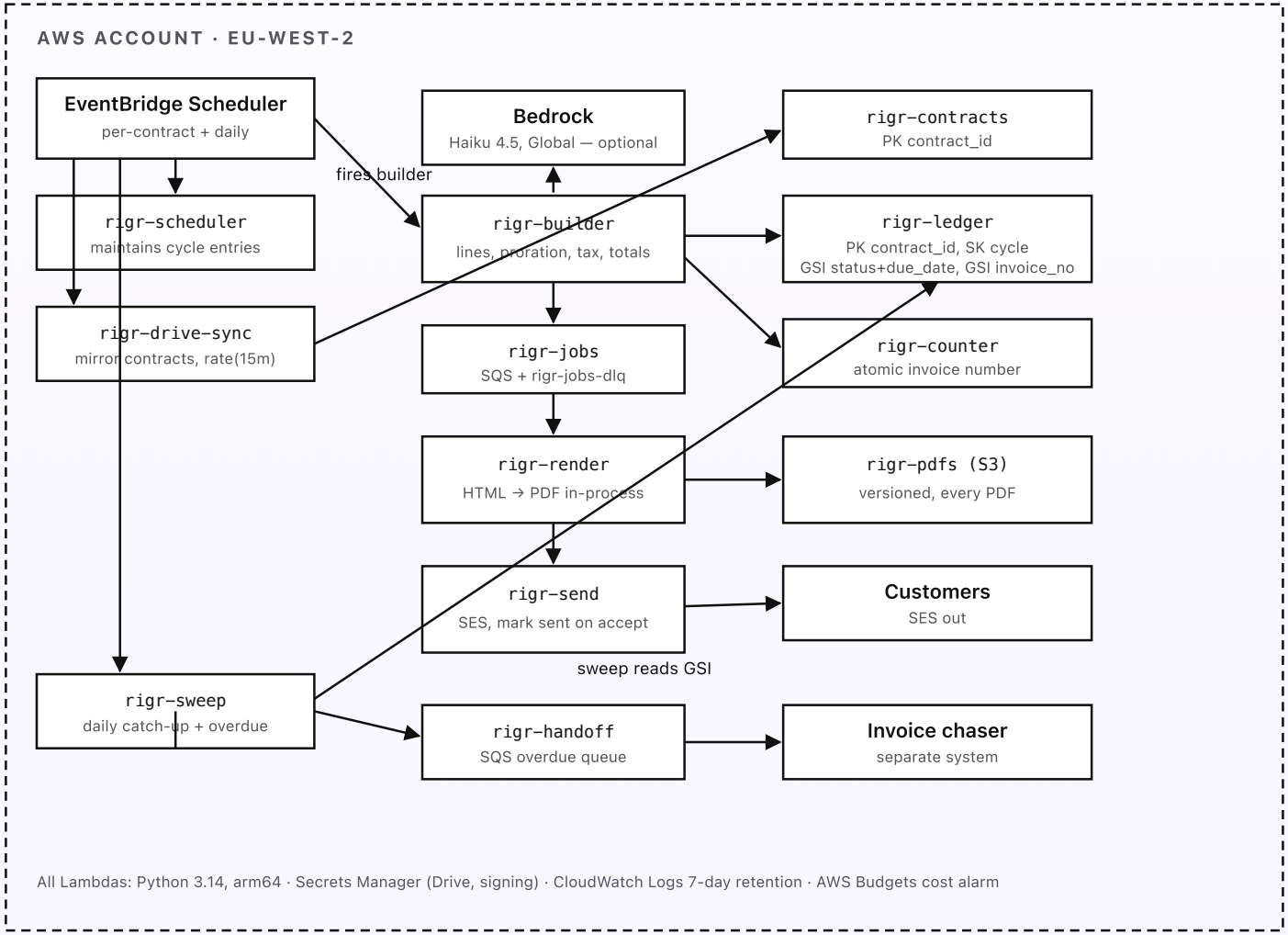

AWS ACCOUNT · EU-WEST-2
EventBridge Scheduler
per-contract + daily
Bedrock
Haiku 4.5, Global — optional
rigr-contracts
PK contract_id
fires builder
rigr-scheduler
maintains cycle entries
rigr-builder
lines, proration, tax, totals
rigr-ledger
PK contract_id, SK cycle
GSI status+due_date, GSI invoice_no
rigr-drive-sync
mirror contracts, rate(15m)
rigr-jobs
SQS + rigr-jobs-dlq
rigr-counter
atomic invoice number
rigr-render
HTML → PDF in-process
rigr-pdfs (S3)
versioned, every PDF
rigr-send
SES, mark sent on accept
Customers
SES out
sweep reads GSI
rigr-sweep
daily catch-up + overdue
rigr-handoff
SQS overdue queue
Invoice chaser
separate system
All Lambdas: Python 3.14, arm64 · Secrets Manager (Drive, signing) · CloudWatch Logs 7-day retention · AWS Budgets cost alarm

Fig 7. The recurring invoice generator drawn for engineers: EventBridge Scheduler driving six Lambdas, an SQS-buffered render-and-send chain, three DynamoDB tables, a versioned PDF bucket, SES out, and a hand-off queue to a separate chaser. One region, one account, no API Gateway.

Lambda functions

Six functions, all Python 3.14 on arm64, all with CloudWatch Logs at 7-day retention. Each does one job; the SQS queue (`rigr-jobs`, with `rigr-jobs-dlq` as its dead-letter queue after five attempts) decouples the build from the slower render-and-send.

- `rigr-drive-sync` — scheduled at `rate(15 minutes)`. Pulls the contracts sheet and rules doc from Drive, writes a snapshot, and upserts rows into `rigr-contracts`. On a new or changed contract it calls `rigr-scheduler` to create or adjust the cycle entry.
- `rigr-scheduler` — maintains the per-contract EventBridge Scheduler entries: creates one when a contract appears, updates it when the cycle or anchor changes, and reschedules it for the next due date after each build. It writes no invoice data.
- `rigr-builder` — the cycle target. Loads the contract, gathers lines (including prorated catch-ups), draws the next number from `rigr-counter`, computes proration, tax and totals in plain Python, optionally calls Bedrock for description text, writes the invoice to `rigr-ledger` with a conditional put on `contract_id` + `cycle`, and enqueues a render job.

- `rigr-render` — SQS-triggered. Reads the ledger record, renders the HTML invoice template to a PDF in-process, and writes the object to `rigr-pdfs`.
- `rigr-send` — emails the PDF via SES with the covering note and view link, then updates the ledger record to *sent* with the SES message id — only on acceptance.
- `rigr-sweep` — scheduled daily. Two jobs: build any invoice whose due date passed without a ledger record (the catch-up), and mark any sent-but-unpaid invoice past due as *overdue*, pushing it onto `rigr-handoff` for the chaser.

Data stores, schedules, and mail

- **DynamoDB (all on-demand).** `rigr-contracts` — PK `contract_id`, the mirrored contract with line items, cycle, anchor, term, and jurisdiction. `rigr-ledger` — PK `contract_id`, SK `cycle` (the billing period), holding the invoice, its status, its due date, and the full workings; a GSI on `status` + `due_date` drives the sweep, and a GSI on `invoice_no` gives direct lookup by number. The `contract_id` + `cycle` key is what makes the build idempotent. `rigr-counter` — a single item incremented with an atomic `UpdateItem` that returns the new value, the source of gap-free invoice numbers.
- **S3.** `rigr-pdfs` — versioning enabled, one object per invoice keyed by number; a reissue writes a new version rather than overwriting, preserving exactly what each customer received.
- **SES.** Verified sending domain with DKIM, used outbound only — there is no inbound mail in this system. Each send carries the rendered PDF and a templated covering message.

- **EventBridge Scheduler.** One entry per contract (grouped) targeting `rigr-builder` on the contract's due date and self-rescheduled after each build; one `rate(15 minutes)` entry for `rigr-drive-sync`; one daily `cron` entry (early morning) for `rigr-sweep`.
- **SQS.** `rigr-jobs` (+ `rigr-jobs-dlq`) buffers render-and-send; `rigr-handoff` is the one-way queue to the separate invoice chaser.
- **Secrets Manager.** Two secrets — the Drive service-account key and the view-link signing key; fetched at call time, never in env vars or the sheet.
- **Bedrock.** Model id `anthropic.claude-haiku-4-5` via the Global cross-Region inference profile, invoked only by `rigr-builder`, and only for line descriptions — never a figure.

IAM scope and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `rigr-drive-sync` can read the Drive secret, write `rigr-contracts`, and invoke `rigr-scheduler`; it cannot touch the ledger. `rigr-scheduler` can manage Scheduler entries in its group and nothing else — no table access at all. `rigr-builder` can read `rigr-contracts`, increment `rigr-counter`, write `rigr-ledger`, send to `rigr-jobs`, and is the only role with `bedrock:InvokeModel`, scoped to the one Haiku profile; it cannot send mail. `rigr-render` can read the ledger and write `rigr-pdfs` — no SES, no Bedrock. `rigr-send` can read a PDF, send via SES, and update a ledger record's status, but cannot delete from any table. `rigr-sweep` can read the ledger GSI, update status, and send to `rigr-handoff` — nothing more. Everything runs in `eu-west-2`; the only cross-Region path is Bedrock's Global inference profile, which routes

the optional description call for capacity and is not a data store. An AWS Budgets alarm watches the monthly spend and notifies if it drifts above a few dollars — the early signal that a schedule or a render loop is misbehaving.

DESIGN RULES THAT SHAPED THE BUILD

- One job per function. Six small Lambdas beat one that does everything; the queue decouples render and send from the build.
- No public surface. There is no API Gateway and no inbound mail — everything is driven by Scheduler and SQS.
- Least privilege, per role. Only the builder can call Bedrock and increment the counter; only send can use SES.
- Idempotency at the database. The `contract_id` + `cycle` key and a conditional put make double-billing impossible.
- One region, one optional model. `eu-west-2` throughout; Bedrock Haiku 4.5 via Global inference, called only to polish words.
- A budget alarm is a smoke detector. The cheapest way to learn a schedule looped is a Budgets alert.