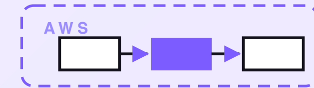
7-PART SERIES · FREE COMPANION



Delivery route planner

Every morning a small delivery or service business has the same puzzle: a list of stops, one or two vans, and no good way to put them in order beyond a driver's memory and a paper map. The result is doubled-back miles, stops missed inside their time window, and customers ringing to ask when the van will turn up. This is the design of a small serverless planner that gathers the day's stops, geocodes the addresses, works out a sensible order — fewest miles, time windows honoured, the van not overloaded — and hands the driver a clean manifest while telling each customer a real ETA window that updates as the round runs. The optimising is plain code; a model only drafts the customer wording, and the plan is always a proposal the driver can override. 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

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

shop.allanninal.dev/w/delivery-route-planner

CONTENTS

Delivery route planner

01 A delivery route planner on AWS for a few dollars a month
02 How the day’s stops get gathered
03 How a route gets optimized
04 How a driver manifest gets built
05 How customer ETAs get sent
06 What the delivery route planner costs
07 Engineering reference: the delivery route planner architecture

PART 1 OF 7

JUNE 29, 2026 PART 1 OF 7 · DELIVERY ROUTE PLANNER SERIES ~10 MIN READ

A delivery route planner on AWS for a few dollars a month

Every morning a delivery or service round starts with the same unglamorous puzzle: here are today's stops, here's the van, what order do we do them in? Most small teams solve it with a driver's instinct and a paper list, which is fine until there are thirty stops, three of them with fixed time windows, and one customer ringing to ask where the van is. This post walks through the design of a small planner that turns the day's stops into an optimised route, hands the driver a clean manifest, and tells each customer when to expect the van — without ever taking the wheel.

KEY TAKEAWAYS

- Each morning the planner gathers the day's stops, geocodes the addresses, and works out a sensible order.
- The optimiser is plain code — a distance matrix, a nearest-neighbour seed, and 2-opt — with no model on the route.
- Time windows and the van's capacity are hard constraints: a stop that won't fit either is never forced in.
- The driver gets an ordered manifest with access notes; each customer gets an ETA window that updates as stops are done.
- Designed on AWS for about $3.10/month at roughly 40 stops a day. The route is a proposal — the driver stays in charge.

The whole system on one page

Before any code, here's the shape of what we're designing. A delivery or service business starts every day with the same puzzle — a list of stops, one or two vans, and a finite number of daylight hours — and solves it the same way: someone eyeballs the addresses and picks an order that feels about right. That works at five stops and falls apart at thirty, especially once a few of them have to happen inside a window. The system below does that ordering properly, the moment the day's list is settled, and tells everyone — driver and customers — what to expect.

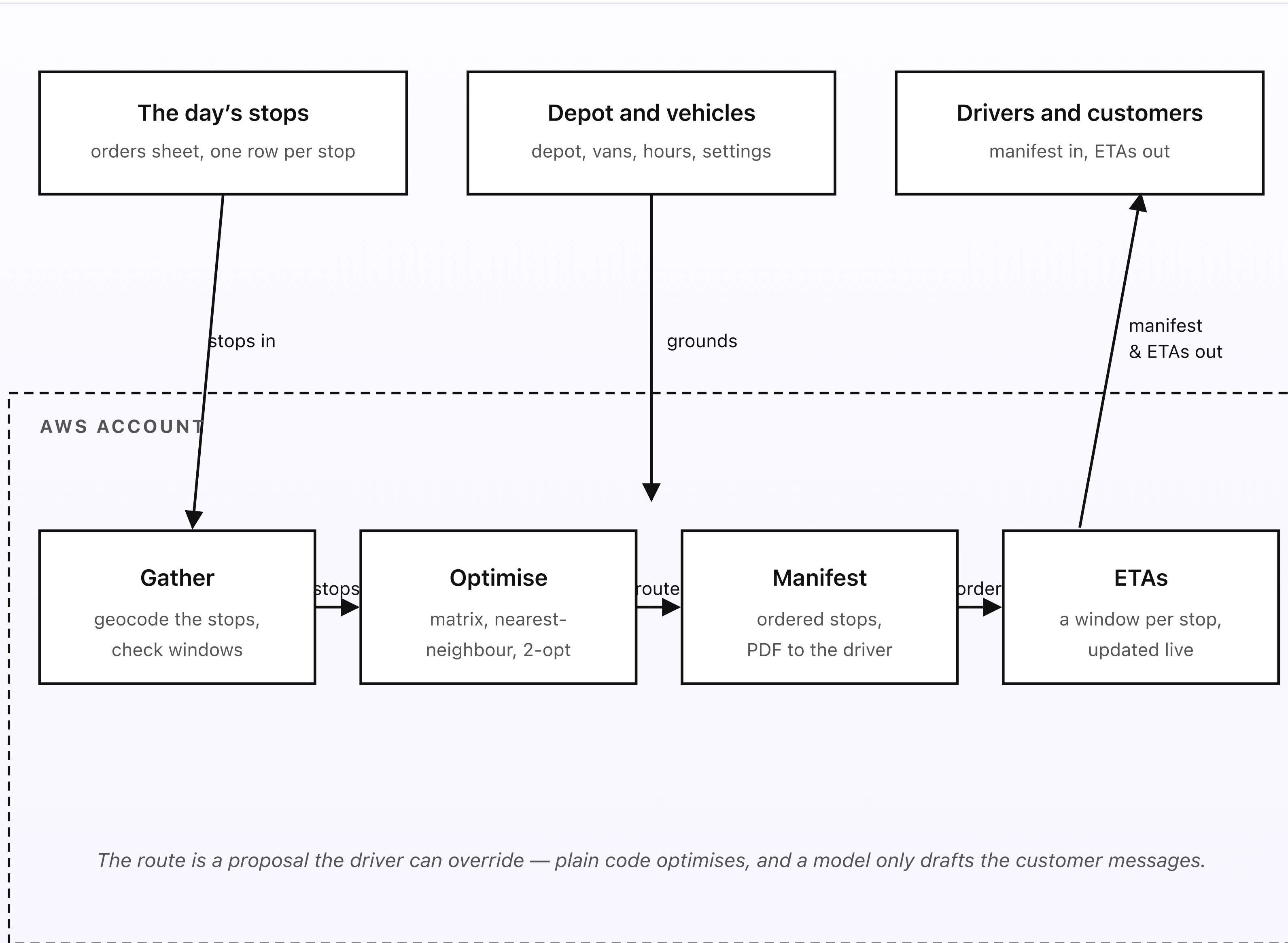

The day's stops
orders sheet, one row per stop
Depot and vehicles
depot, vans, hours, settings
Drivers and customers
manifest in, ETAs out
stops in
grounds
manifest
& ETAs out
AWS ACCOUNT
Gather
geocode the stops,
check windows
stops
Optimise
matrix, nearest-
neighbour, 2-opt
route
Manifest
ordered stops,
PDF to the driver
order
ETAs
a window per stop,
updated live
The route is a proposal the driver can override — plain code optimises, and a model only drafts the customer messages.

Fig 1. Three things outside, four pieces inside AWS. The day's stops flow in; the depot and vehicle settings ground every calculation. Gather geocodes and checks the stops, Optimise orders them, Manifest hands the driver a list, and ETAs keep each customer informed.

What you set up once (the outside)

- **The day's stops.** A Google Sheet in a Drive folder (or your existing orders system) with one row per stop: a stop reference, the customer name, the delivery address, a phone or email for the ETA, an optional time window ("before noon", "2–4pm"), the size or weight — or for a service round, the minutes the job takes — and a free-text access note ("side gate, dog"). You already produce most of this in the normal course of taking orders; this just keeps it where the planner can read it each morning. Part 2 covers exactly how it's gathered.
- **Depot and vehicles.** One short settings doc in the same folder. It holds the depot address every route starts and ends at, the vans and what each can carry (boxes, weight, or pallet count), the working day ("08:00 to 17:00"), the average time a driver spends at a stop, and which geocoding and distance-matrix provider to use. Changing the cut-off time or adding a second van is an edit here — not a deploy.
- **Drivers and customers.** The driver opens a link on their phone and sees the manifest — the stops in order, each with its address, window, and access note. The customer gets a single message: "Your delivery is due between 10:30 and 11:30 today," updated if the round runs ahead or behind. Neither side ever sees the optimiser's internals; they see a list and a window.

What runs every morning (the inside)

- **Gather.** A scheduled job wakes at a set time, reads the day's stops from the mirror, and turns each address into a latitude and longitude. Addresses repeat — the same streets come up week after week — so a cache means most lookups cost nothing. It also validates each row: a stop with no usable address, an impossible window, or a load bigger than any van is pulled out and flagged rather than quietly breaking the route. This is Part 2.
- **Optimise.** With clean coordinates in hand, plain Python builds a matrix of travel times between the depot and every stop, seeds a route by nearest-neighbour (start at the depot, always go to the closest stop not yet visited), then runs 2-opt to uncross any legs that double back. Time windows and van capacity are checked on every candidate route; anything that would make the van late or overloaded is rejected outright. No model touches this. This is Part 3.
- **Manifest.** The ordered route is rendered into a driver manifest — stop 1, stop 2, stop 3, each with the address, the window, and the access note — written once to a PDF and dropped in S3, then sent to the driver as a link. The driver can re-order or skip; the manifest is the suggested order, not a locked instruction. This is Part 4.
- **ETAs.** As the round is planned, each customer gets an arrival window for their stop, the wording drafted by one small Bedrock call. As the driver checks stops off, those events flow back in and the remaining windows are recomputed from where the van actually is, so a slow morning slides everyone's ETA without anyone retyping a thing. This is Part 5.

In plain words

At 6am a florist's sheet has 34 deliveries for the day, six of them flagged "before noon" for offices that lock up at lunch. The planner geocodes the addresses — 31 are already in the cache from previous weeks, so only three are looked up — and orders the round so the six timed stops fall in the morning, the van loops out and back without crossing its own path, and the heaviest crates come off first. The driver opens the manifest at 7:40: 34 stops, in order, each with the postcode, the window, and notes like "reception, ask for Priya." Every customer has already had a text: "Your flowers are due between 9:15 and 10:15 today." When the driver hits traffic and falls twenty minutes behind by stop 12, the afternoon windows quietly shift and the next few customers get an updated text — no one rings to ask.

The cost of running this is about $3.10 a month at that volume. The cost of *not* running it is the doubled-back miles nobody measures, the "before noon" stop reached at 12:20, and the steady trickle of "where's my delivery?" calls that land on whoever is nearest the phone.

DESIGN RULES THAT SHAPED EVERY DECISION

- The plan is a proposal. The manifest is the suggested order; the driver can re-order, skip, or insert a stop at any time.
- Plain code optimises. The matrix, nearest-neighbour, and 2-opt are deterministic Python — no model decides the route.
- Windows and capacity have a veto. A route that arrives late or overloads a van is rejected, never “optimised” into existence.
- The model only writes words. Bedrock drafts the customer ETA message and nothing else; swap it for a template and lose nothing structural.
- Settings live in a doc. Depot, vans, hours, and the average stop time change without a deploy.
- Pay per morning, not per hour. Nothing runs when there are no stops to plan, so the idle bill is zero.

Why this shape

Most small rounds are planned one of three ways: the driver orders the stops from memory, someone senior spends twenty minutes with a map before the vans roll, or there’s an expensive fleet product that assumes a depot of fifty vehicles and bills accordingly. Memory is fine until the new driver doesn’t know the shortcuts. The twenty-minute map is the first thing dropped on a busy morning, and it can’t react when stop 12 runs late. And the fleet product solves a problem a two-van florist doesn’t have, at a price that doesn’t fit.

The shape above keeps the orders sheet you already maintain as the list of stops, leans on a hosted matrix service (or a simple distance estimate) for the travel times, and adds a small system that does the ordering properly every morning for the cost of a coffee. The optimisation that people assume needs AI is the boring, well-understood part — nearest-neighbour and 2-opt have been ordering routes for decades — so there's no model on the path that matters, and the one place a model does appear, the customer text, can be turned off without breaking anything.

The next four posts walk through each piece in turn: how the day's stops get gathered, how a route gets optimised, how a driver manifest gets built, and how customer ETAs get sent. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 29, 2026 PART 2 OF 7 · DELIVERY ROUTE PLANNER SERIES ~7 MIN READ

How the day's stops get gathered

Before anything can be put in order, the system has to know exactly where each stop is and what it demands. This post is about that first step alone: how the morning build gathers the day's stops from the place you already keep them, turns each address into a real coordinate without paying to geocode the same street twice, and throws out the bad rows before they can poison the route.

KEY TAKEAWAYS

- A scheduled morning job reads the day's stops from the orders mirror — no one has to press a button.
- Each address is turned into a coordinate; a geocode cache means repeat streets are looked up once, not daily.
- Every row is validated — address, time window, and size — and bad ones are pulled out and flagged, not routed.
- The output is a clean, geocoded stop list the optimiser can trust; nothing downstream has to guess.
- The cut-off time, the cache, and the validation rules are settings — changing them never needs a deploy.

The morning build

Everything starts with a clock. At a time you set — say 06:00, well before the vans load — EventBridge Scheduler fires the gather function. It doesn't wait for anyone to open a laptop; by the time the first driver arrives, the day's round is already planned. The job's only purpose is to produce one thing the rest of the system can rely on: a clean list of stops, each with a real coordinate, each known to fit a van and a window. Garbage here becomes a broken route three steps later, so the gather is deliberately fussy.

It reads the day's stops from the mirror — the orders sheet, copied into AWS every few minutes by a small sync function so the planner never depends on a live

Google call at 6am. From the settings doc it picks up the depot address, the working hours, and the average time spent at a stop, because the optimiser will need all three. Then it works through the rows one at a time.

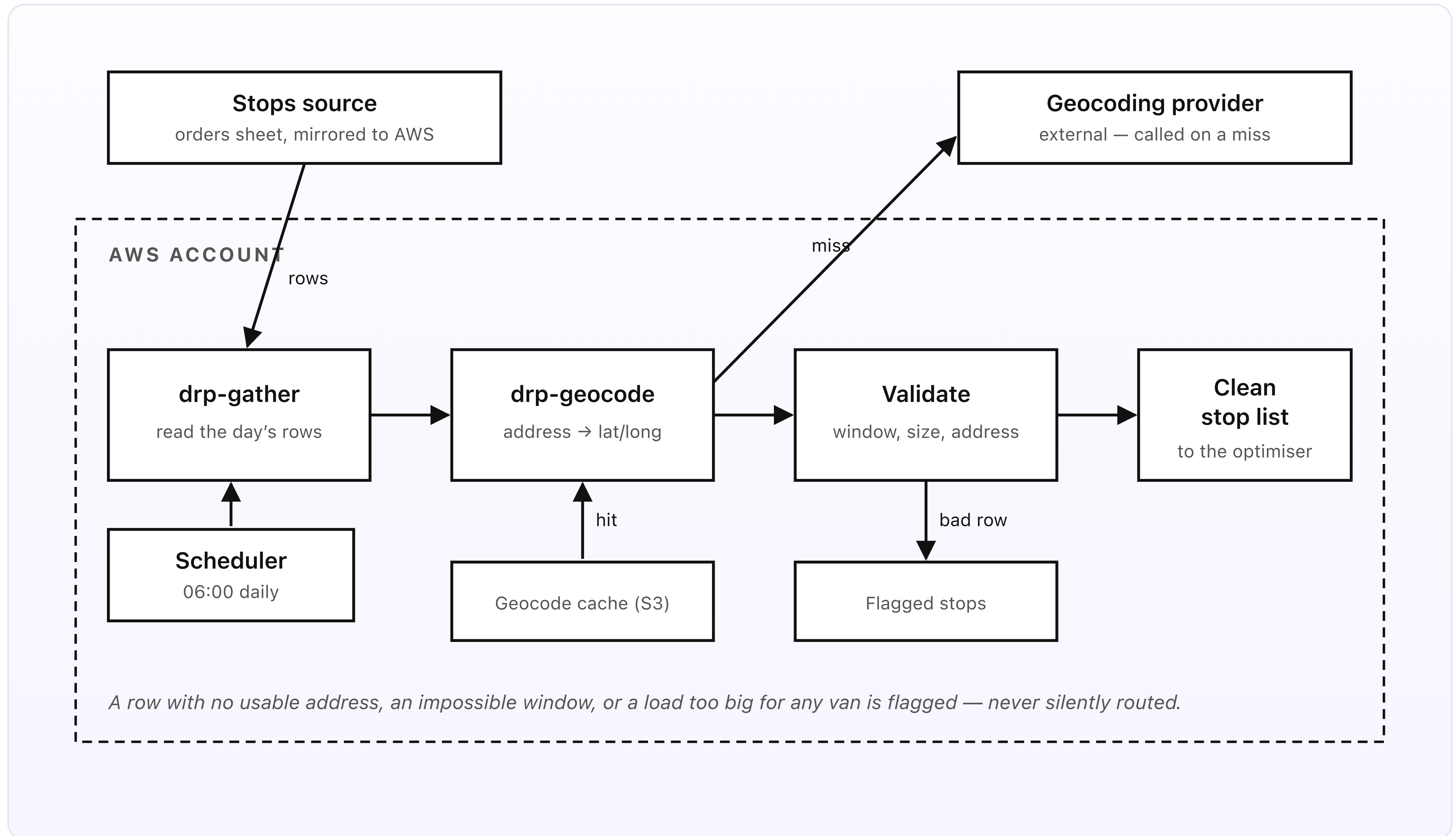


Fig 2. The morning gather. A schedule wakes the job; it reads the rows, geocodes each address against a cache (calling the provider only on a miss), validates window and size, and hands a clean, coordinate-tagged list to the optimiser. Bad rows drop out to a flagged list.

Turning addresses into coordinates

The optimiser thinks in coordinates, not in "14 Mill Lane." So every stop's address has to become a latitude and longitude, and that's what geocoding does. The naive version calls a geocoding API for every stop, every morning — which means paying to look up the same regular customer's address five times a week. The planner doesn't do that.

Instead, geocoding goes through a cache. The address is normalised (trimmed, upper-cased, punctuation stripped) into a stable key, and that key is looked up first in an S3-backed cache. A hit returns the stored coordinate for nothing; only a miss — a genuinely new address — calls the external provider, using an API key kept in Secrets Manager, and the result is written back to the cache for next time. On a typical round most stops are regulars or repeat streets, so the cache hit rate sits high and the provider is called only a handful of times a day. That single decision is what keeps the system's biggest variable cost small, and Part 6 returns to it with numbers.

A coordinate that comes back as low-confidence — the provider matched the town but not the street — isn't silently trusted. It's treated like any other bad row: pulled out and flagged, so a person can fix the address rather than have the van sent to the middle of a postcode district.

Throwing out the bad rows

Validation is the unglamorous half of this step and the half that earns its keep. Each row is checked against three things before it's allowed into the route:

- **A usable address.** Blank, obviously incomplete, or geocoded to low confidence — flagged. There's no point routing a van to a place the system can't locate.
- **A sane time window.** A window that has already passed, that's narrower than the time it takes to do the stop, or that falls outside the working day — flagged. A "deliver 06:00–06:30" on a round that starts at 08:00 is a mistake, not a constraint.
- **A load that fits.** A stop whose size or weight is larger than any single van can carry — flagged. The optimiser can split a day across vans, but it can't split one stop that no vehicle can hold.

Flagged rows don't vanish. They're written to a flagged-stops list with the exact reason — "address not found," "window already past," "exceeds van capacity" — and surfaced to whoever runs the round, so the fix happens before the vans leave rather than as a confused driver phoning in from a lay-by. Everything that passes all three checks becomes a clean stop record — reference, coordinate, window, size, access note — written to DynamoDB and handed to the optimiser. From here on, nothing downstream has to wonder whether a stop is real.

WHY THIS SHAPE

- Plan before anyone arrives. A morning schedule means the round is ready when the first driver walks in.
- Geocode once, not daily. The cache turns repeat streets into free lookups and keeps the biggest variable cost down.
- Validate hard, fail loud. Bad rows are flagged with a reason, never routed into a van's afternoon.
- Read from a mirror, not a live call. The orders sheet is copied into AWS so 6am never depends on a third-party API being up.
- Hand on a clean list. Everything downstream trusts that a stop has a real coordinate and fits a window and a van.

PART 3 OF 7

JUNE 29, 2026 PART 3 OF 7 · DELIVERY ROUTE PLANNER SERIES ~8 MIN READ

How a route gets optimized

This is the heart of the system, and the part people assume needs clever AI — it doesn't. This post is about the optimiser: how a flat list of geocoded stops becomes an ordered route with plain code, why nearest-neighbour then 2-opt is the right tool for a few dozen stops, and what 'optimised' honestly means when time windows and a van's capacity get a veto.

KEY TAKEAWAYS

- First build a matrix of travel times between the depot and every stop — from a hosted service, or a straight-line estimate.
- Seed a route with nearest-neighbour: start at the depot, always hop to the closest stop not yet visited.
- Improve it with 2-opt: repeatedly find two legs that cross and reverse the bit between them to uncross the route.
- Time windows and van capacity are hard constraints — a swap that breaks either is rejected, not accepted because it's shorter.
- It's a strong, fast heuristic for dozens of stops — reliably close to best, in milliseconds — not a proof of the shortest route.

First, a matrix

You can't order stops without knowing the cost of going from each one to each other one. So the first thing the optimiser builds is a matrix: for every pair of points — the depot and all the stops — how long does it take to drive between them? With n stops plus the depot, that's an $(n+1) \times (n+1)$ grid of travel times.

There are two honest ways to fill it. The accurate way is a hosted distance-matrix service, which knows real roads, one-way streets, and typical traffic; you send it the coordinates and it returns the grid. The cheap way is to compute straight-line (haversine) distances between coordinates and multiply by a fudge factor for the fact that roads aren't straight. The hosted matrix gives a better route and costs per call; the straight-line estimate is free and good enough for dense urban rounds where everything is close together. The planner uses whichever the settings doc names, and caches the matrix for the day so the same grid isn't bought twice. Either way, once the matrix exists, the rest of the optimiser never touches a map again — it just reads numbers from the grid.

A first guess: nearest-neighbour

With the matrix in hand, the optimiser needs a starting route — any complete, valid order it can then improve. The simplest sensible one is nearest-neighbour: start at the depot, look at every stop not yet visited, go to the closest one, and repeat until they're all done, then return to the depot. It's greedy — it only ever looks at the next hop — and it's fast, and it produces a route that's roughly right but usually has a few ugly legs near the end, where the only unvisited stops left are back the way you came.

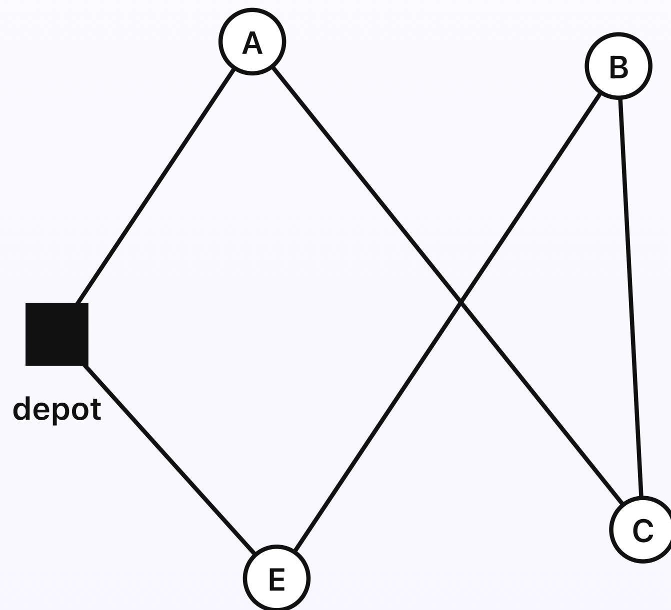
Nearest-neighbour isn't the answer; it's the first draft. Its job is to give 2-opt something to chew on.

Uncrossing the route: 2-opt

The clearest sign of a bad route is a crossing — two legs of the round that overlap on the map. Whenever a route crosses itself, you can always make it shorter by uncrossing that pair of legs, and that's exactly what 2-opt does. It takes two legs of the current route, reverses the run of stops between them, and checks the matrix: is the new route shorter? If yes, keep it. If no, put it back. Do that for every pair of legs, over and over, until a full pass finds no improving swap. What's left is a route with no crossings and no obvious slack.

Nearest-neighbour seed

~38 miles — A–C and B–E cross



After 2-opt

~33 miles — no crossings

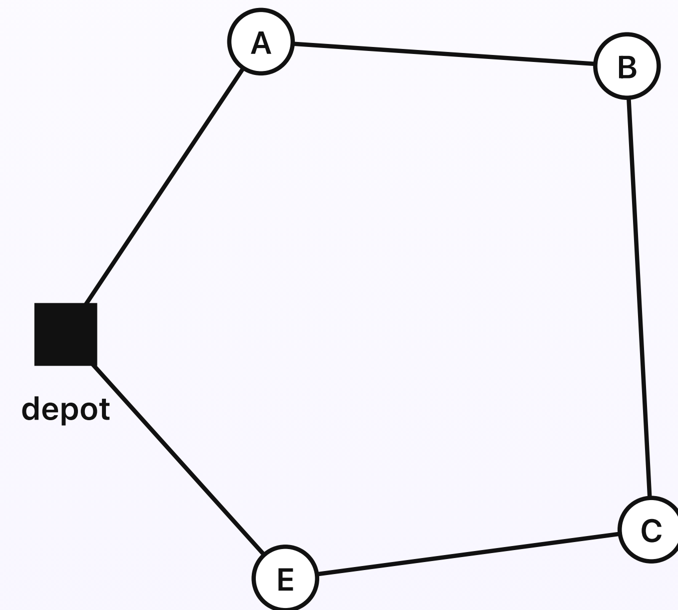


Fig 3. Nearest-neighbour gives a route that crosses itself (left): the leg A–C and the leg B–E overlap. 2-opt reverses the segment between them, visiting A–B–C–E instead (right). The crossing is gone and the round is shorter.

A worked example

Take the five stops above. Nearest-neighbour starts at the depot and goes to A (the closest). From A, the nearest unvisited stop turns out to be C, because the greedy step only looks at the next hop and C is marginally closer than B at that moment. From C it picks B, from B it picks E, and back to the depot. That gives the order **depot → A → C → B → E → depot** — about 38 miles — and you can see the problem on the left panel: the leg from A down to C and the leg from B down to E cross right through the middle of the map. The greedy “always go to the nearest” rule painted itself into that corner near the end.

2-opt spots it. It tries reversing the segment between those two crossing legs — the run C–B becomes B–C — turning the order into **depot → A → B → C → E → depot**. It checks the matrix: the new order is about 33 miles, five miles shorter, so it keeps it. The crossing is gone (right panel), and a second pass finds no further improving swap, so the optimiser stops. Five miles on one small round, every single day, is the whole reason this step exists.

Windows and capacity get a veto

Pure 2-opt only cares about distance. A real round has two extra rules that distance must never be allowed to break, so they’re enforced as *hard constraints*: every candidate route — the nearest-neighbour seed and every 2-opt swap — is checked against them before it can be accepted.

- **Time windows.** The optimiser walks the candidate route forward from the depot, adding travel time and the average time spent at each stop, and asks: does every timed stop get reached inside its window? A swap that shaves a mile

but pushes the “before noon” office to 12:10 is rejected, even though it’s shorter. Shorter is only better when it’s also on time.

- **Capacity.** If a van can’t carry the whole day, the stops are split across vans (or across two trips), and no van’s running load is ever allowed to exceed what it holds. A swap that would overfill a vehicle is rejected the same way.

So the optimiser isn’t minimising distance in the abstract — it’s finding the shortest route *among the ones that are actually allowed*. That distinction is the difference between a clever-looking plan and a plan a driver can really run.

What “optimised” honestly means

It’s worth being straight about this, because the word “optimised” oversells easily. Nearest-neighbour plus 2-opt is a *heuristic*. It does not prove it has found the single shortest possible route — finding that, guaranteed, is the travelling-salesman problem, which gets brutally expensive as stops multiply. What this approach does is reliably get very close, very fast: for the few-dozen-stops range a small business actually runs, it returns a route within a few percent of optimal in milliseconds, on plain Lambda, for nothing. That trade — “almost certainly near-best, instantly and free” over “provably best, slowly and expensively” — is exactly the right one for a two-van florist, and it’s why there is no model anywhere in this step. The maths has been understood for fifty years; it just needed wiring up.

WHY THIS SHAPE

- Build the matrix once. Every later step reads travel times from the grid instead of touching a map.
- Seed, then improve. Nearest-neighbour gives a fast first draft; 2-opt polishes it by uncrossing legs.
- Constraints beat distance. Time windows and capacity have a veto; a shorter route that breaks them is rejected.
- Heuristic, not oracle. Near-best in milliseconds for nothing, not provably-best slowly — the right trade at this scale.
- No model on the route. The whole optimiser is deterministic Python; the same stops always give the same plan.

PART 4 OF 7

JUNE 29, 2026 PART 4 OF 7 · DELIVERY ROUTE PLANNER SERIES ~7 MIN READ

How a driver manifest gets built

An optimal route that lives only in a database helps no one. This post is about the handover to the person who actually does the work: how the ordered route becomes a driver manifest — stop by stop, with the address, the access note, and the window for each — rendered once to a PDF, dropped in S3, and sent as a link the driver opens on their phone. And why every line of it is a suggestion, not an order.

KEY TAKEAWAYS

- The optimised route becomes a driver manifest: stops in order, each with address, window, access note, and a phone number.
- It's rendered once to a PDF, written to S3, and sent to the driver as a link — no app to install.
- Each stop carries a check-off action, so tapping "done" is what later slides everyone else's ETA.
- The manifest is the suggested order, not a locked instruction — the driver can re-order, skip, or insert a stop.
- One manifest per van per day; a fresh plan (a late order, a cancellation) reissues a new version, the old link expires.

From a route to a round sheet

The optimiser's output is a list of stop references in an order — correct, but useless to a human as it stands. A driver doesn't want an array; they want a round sheet they can glance at between stops. The manifest step turns one into the other. It reads the ordered route, joins each reference back to the clean stop record from Part 2 to pull in the human details, and lays them out in sequence: stop 1, stop 2, stop 3, each with the customer name, the full address and postcode, the time window if there is one, the access note, and a tap-to-call phone number. At the top sits the depot and the planned start time; at the bottom, the return.

It's rendered once, in the morning, to a PDF — a format that opens on any phone, prints if the driver prefers paper, and looks the same offline in a dead-signal lane as it does on wifi. The PDF is written to S3 and the driver is sent a link to it, by SES email or an SMS through SNS, whichever the settings doc names. No app to install, no login to forget; a link, a list, and a phone.

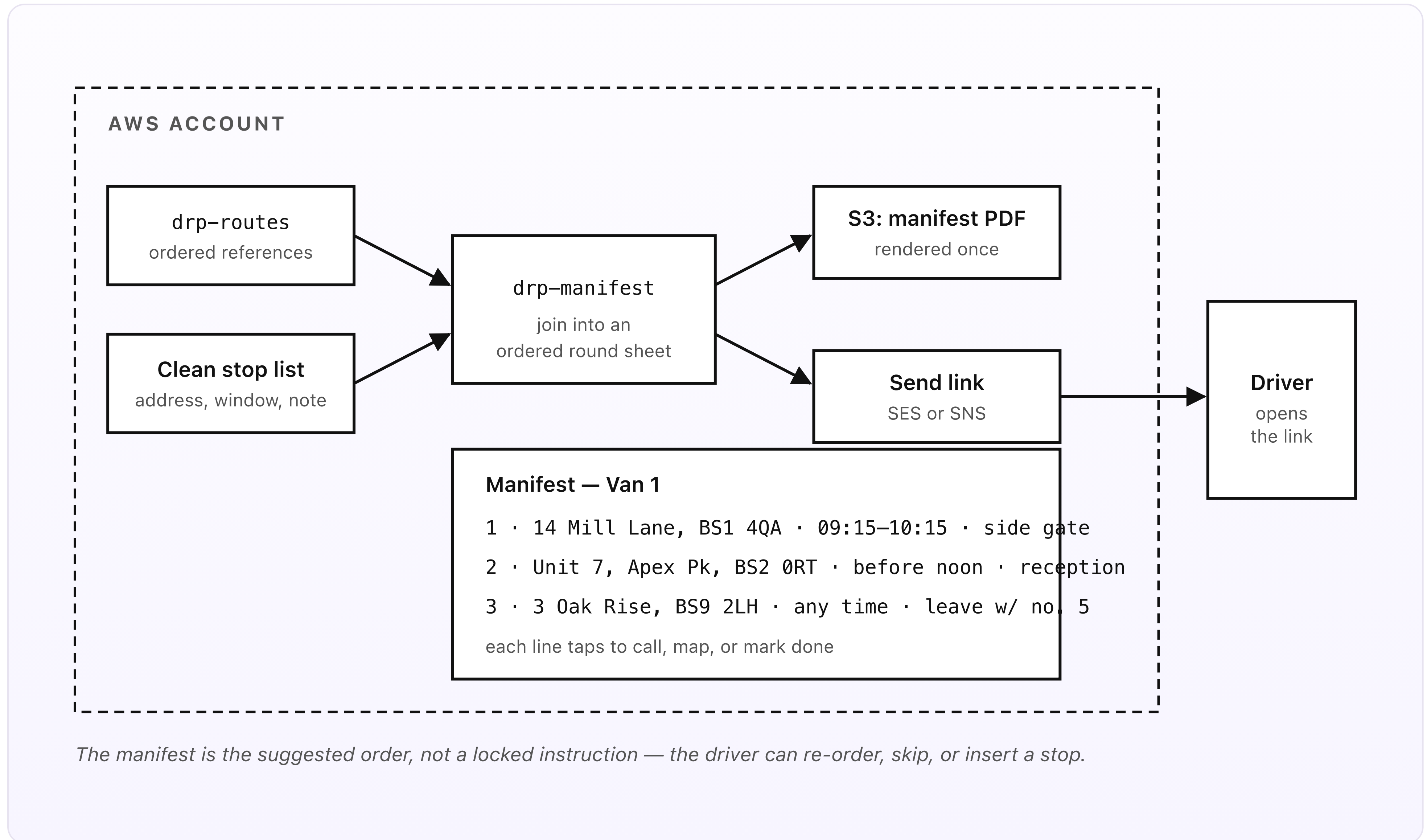


Fig 4. Building the manifest. The ordered route is joined to each stop's human details, rendered once to a PDF in S3, and sent to the driver as a link. The driver sees a plain round sheet — address, window, access note — with each line tappable.

What goes on each line

A manifest line is built for someone glancing at it through a windscreen, so it carries only what the next stop needs and nothing else:

- **The position and address.** “Stop 7 of 34” and the full address with postcode, so the driver knows where they are in the day and exactly where to point the van. The postcode doubles as a tap-to-navigate link into whatever map app the phone already has.
- **The window, if any.** Shown plainly — “before noon,” “2–4pm” — so the driver can see at a glance which stops are flexible and which aren’t. The optimiser already guaranteed the order respects them; the line just reminds the human.
- **The access note.** The single most undervalued field on the sheet. “Side gate, dog,” “ask for Priya at reception,” “leave with number 5 if out” — the local knowledge that turns a failed delivery into a successful one. It comes straight from the stop record and is shown verbatim.
- **A check-off action.** Each line can be marked done. That tap is not cosmetic: it’s the event that tells the rest of the system the van has left a stop, which is what keeps the downstream ETAs honest in Part 5.

A proposal, not an instruction

This is the post where the series’ central guardrail becomes concrete. The manifest is the order the optimiser thinks is best — but the driver, not the planner, runs the round. A regular customer flags the van down for an unplanned collection; a road is shut; the driver simply knows the area better than the matrix does. In every case they re-order, skip, or insert a stop, and the system follows

their actual progress rather than nagging them back onto the original sequence. Marking stop 9 done before stop 8 isn't an error the system corrects; it's information the system accepts. The plan exists to save the driver thinking on a normal day, not to override their judgement on an abnormal one.

There is exactly one manifest per van per day, versioned. If the day's list genuinely changes before the vans leave — a late order lands, a customer cancels — the morning build can be re-run, the optimiser re-orders, and a fresh manifest version is issued; the old link stops working so no one drives yesterday's plan by accident. Once the van is rolling, though, the manifest is fixed and changes are the driver's, captured through their check-offs rather than through a reshuffled sheet.

WHY THIS SHAPE

- Built for a glance. Each line shows position, address, window, and access note — nothing the driver doesn't need mid-round.
- A link, not an app. A PDF in S3 opens on any phone, prints, and works offline; nothing to install or log into.
- The access note earns its place. Local knowledge, shown verbatim, is what turns a failed drop into a completed one.
- Check-off is an event. Marking a stop done is what later moves everyone else's ETA, not just a tick on a list.
- One versioned manifest per van. A real change reissues it and expires the old link; en route, the driver is in charge.

PART 5 OF 7

JUNE 29, 2026 PART 5 OF 7 · DELIVERY ROUTE PLANNER SERIES ~7 MIN READ

How customer ETAs get sent

The route saves the business miles; the ETAs save the customer a wasted afternoon by the door. This post is about the outbound side: how the planner sends each customer a sensible arrival window, the one place a model is allowed near the system, and how a driver tapping ‘done’ on one stop quietly slides everyone else’s ETA earlier or later so the windows stay true as the day runs.

KEY TAKEAWAYS

- Each customer gets one arrival window for their stop — “due between 10:30 and 11:30” — not a raw clock time.
- The window is computed from the route and the average stop time; a model only turns the numbers into a friendly sentence.
- Driver check-offs flow back through EventBridge, and the remaining ETAs are recomputed from where the van actually is.
- ETAs go out by SES email for free, or SMS through SNS where the per-message carrier fee is worth it.
- It only ever sends a window. It can’t reschedule or cancel; a slipping round triggers an update, not a decision.

A window, not a time

The route already knows, near enough, when the van reaches each stop: walk the order forward from the depot start time, add the travel time from the matrix and the average minutes spent at every earlier stop, and you have a planned arrival for stop *n*. The planner never tells a customer that exact minute, though, because no round survives contact with traffic and a single missing parking space. It tells them a window — the planned arrival, widened by a sensible buffer on each side — so “arrives 10:52” becomes “due between 10:30 and 11:30.” A window the van comfortably hits beats a precise time it misses by four minutes; the buffer is the difference between a customer who trusts the messages and one who stops reading them.

The numbers — the window, the stop position, the customer's name and address — are all computed by the same plain code that built the route. The only thing left is to say it like a person. That's the one job handed to a model.

The one place a model runs

For each customer, a single Bedrock Claude Haiku 4.5 call is given the already-computed facts — business name, customer name, the window, and whether this is the first notice or an update — and asked to write one short, friendly message in the business's voice. It is told, firmly, to use only those facts: invent no times, promise no specifics beyond the window, add no apology the business didn't ask for. The model never sees the route, never computes a time, and never decides who gets a message; it dresses numbers it was handed. If you'd rather not run a model at all, the same facts drop straight into a fixed template — "Hi {name}, your delivery from {business} is due between {from} and {to} today" — and the system works identically, a little less warmly. The model is a convenience on the wording, never a dependency on the logic.

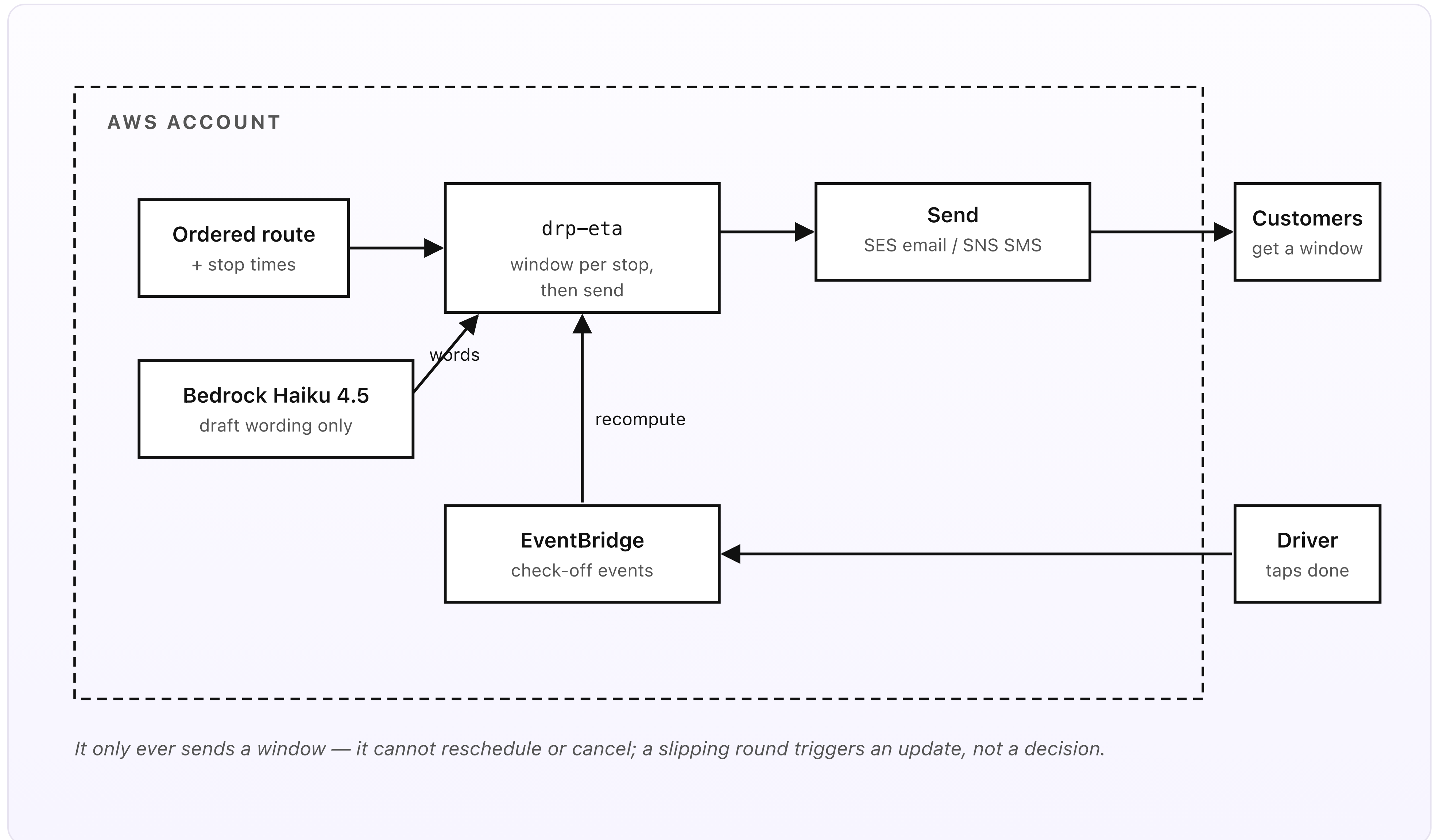


Fig 5. ETAs out, and how they stay true. `drp-eta` computes a window per stop and Bedrock writes the sentence; SES or SNS sends it. When the driver checks a stop off, the event flows through EventBridge and the remaining windows are recomputed from

where the van actually is.

Keeping the window honest

A window sent at 6am and never touched again is a lie by lunchtime. The planner keeps it honest with the check-offs from Part 4. Every time the driver marks a stop done, that tap becomes an EventBridge event — stop, van, the actual time it was completed — and the ETA function picks it up. It now knows two things the morning plan only guessed: which stops are genuinely finished, and what time it really is. So it re-walks the *remaining* stops from the van's current position and the current clock, producing fresh windows for everyone still waiting.

Most of the time the change is small and no message goes out — a window that moves four minutes isn't worth a buzz in someone's pocket. But when the round has slipped enough that a later customer's window has genuinely moved — the van is half an hour down after a bad stretch — that customer gets one updated message: "running a little behind; your delivery is now due between 14:15 and 15:15." The threshold for sending an update is a setting, so a business can choose how chatty it wants to be. The customer who took the morning off to wait in finds out at 1pm that it'll be after lunch — from a text, not from staring out of the window.

Email is cheap; SMS when it counts

Two channels carry the messages. SES email is effectively free and fine for the many customers who'll be near a phone anyway. SMS through SNS lands more reliably and gets read faster, which matters for a tight window or a customer who

has to physically be there to receive — but each text carries a per-message carrier fee that sits outside the AWS bill (Part 6 is careful about this). So the channel is a per-customer choice driven by the settings doc and the data on the stop: a flagged “text me” preference, or a stop with a narrow window, gets an SMS; the rest get email. Either way it’s one message per customer per change, never a stream.

And the boundary is firm: the ETA side only ever *tells*. It can’t move an appointment, can’t cancel a stop, can’t promise a redelivery. If a customer replies “I won’t be in,” that’s a human’s job to handle — the reply lands with whoever runs the round, with the stop and the round attached. A slipping route makes the planner send a new window; it never makes the planner make a decision.

WHY THIS SHAPE

- Send a window, not a minute. A buffered window the van hits keeps customers trusting the messages.
- The model only writes. Bedrock turns computed numbers into a sentence; swap in a template and lose nothing structural.
- Check-offs keep it true. Each “done” event recomputes the remaining windows from where the van really is.
- Quiet unless it matters. A window moves silently until it crosses a threshold; then one update goes out, never a stream.
- Tell, never decide. The ETA side can’t reschedule or cancel; anything that needs a choice goes to a person.

PART 6 OF 7

JUNE 29, 2026 PART 6 OF 7 · DELIVERY ROUTE PLANNER SERIES ~7 MIN READ

What the delivery route planner costs

A planner that costs more than the diesel it saves is a toy. This post is the cost breakdown: every AWS service the system touches, what each adds up to at around 1,000 stops a month, and why the total lands near $3.10 — plus the one line that genuinely scales with use, the geocoding and distance-matrix API, and the SMS fee that sits outside the AWS bill entirely.

KEY TAKEAWAYS

- About $3.10/month at roughly 40 stops a day — around 1,000 a month — and the idle cost is essentially zero.
- The biggest line is the geocoding and distance-matrix API; the cache is what keeps it from being much bigger.
- Bedrock is the next line — one small Haiku call per customer message — and it's optional.
- Secrets Manager is the one true fixed cost: $0.40 a month for the routing API key, whether you plan one stop or a thousand.
- Customer SMS carries a per-message carrier fee that sits outside the AWS bill entirely — email ETAs are free.

Where the money goes

The planner is serverless end to end. Nothing runs between this morning's build and tomorrow's; there's no instance idling overnight and no idle bill. You pay for a day only when there are stops to plan. At a typical small-round volume — call it 40 stops a day, around 1,000 a month, one or two vans — here's the whole bill, line by line.

AWS service (and the one external API)	What it does here	Monthly
Geocoding & matrix API	New-address geocodes and the daily distance matrix — cache absorbs the rest	$1.20
Bedrock (Claude Haiku 4.5)	One message-drafting call per customer notice and update	$0.50
Secrets Manager	One secret — the routing/geocoding API key ($0.40 each)	$0.40
DynamoDB (on-demand)	Stops, routes, geocode cache index, driver progress — small reads and writes	$0.25
CloudWatch Logs	Function logs, 7-day retention	$0.20
Lambda (Python 3.14, arm64)	Gather, geocode, optimise, manifest, ETA, check-off, sweep, drive-sync	$0.20
SES (outbound)	Manifest links and customer ETA emails	$0.15

AWS service (and the one external API)	What it does here	Monthly
S3	Manifest PDFs and the geocode cache	$0.10
SQS + DLQ	Buffering between the build steps	$0.05
EventBridge (Scheduler + check-off bus)	The morning build and the driver check-off events	$0.05
AWS Budgets	Cost alarm (first two budgets are free)	$0.00
Total	**~40 stops/day, ~1,000/month**	**$3.10**

The shape of that bill is the point. The geocoding and matrix API — the only genuinely external thing the system buys — is the largest line, and it's still under half the total, because most addresses are already in the cache and the matrix is bought once a day, not once a stop. Bedrock, the one place a model runs, is second and entirely optional. Everything that does the actual work — the optimiser, the manifest builder, the ETA maths — is plain Python on Lambda and rounds to cents, because compute is cheap and the work is small.

Only the routing API and Bedrock scale with use; everything else, bar Secrets Manager, rounds to zero at idle.

Fig 6. The monthly bill at about 1,000 stops. The geocoding and matrix API is the biggest single line; Bedrock is the next and it's optional. The work that orders the round — Lambda, DynamoDB, S3 — rounds to cents.

The main variable, and the one fixed cost

Two lines deserve a closer look because they behave so differently. The geocoding and matrix API is the system's *variable* cost — the line that moves with how you run. It would be the dominant cost if the planner geocoded every address every morning and bought a fresh matrix per stop; the cache and the once-a-day

matrix are precisely the design choices that hold it to about $1.20. Plan more stops, or churn through more new addresses, and this is the line that grows. It's also the line you can shrink to almost nothing by choosing the straight-line distance estimate from Part 3 instead of a hosted matrix, trading a slightly rougher route for a near-free one.

Secrets Manager, by contrast, is the only line that bills while the system sleeps: $0.40 a month for the one secret that holds the routing API key, whether you plan a single stop or ten thousand. It's a small, honest fixed cost — the price of keeping a credential where it belongs rather than baked into a function's environment. Everything else on the table is genuinely usage-priced and rounds to zero at idle, which is exactly what you want from a system that only works for a couple of minutes each morning.

The cost that's not on the table

One real cost is deliberately absent: customer SMS. Sending ETAs by SES email is effectively free and on the table above. Sending them by SMS through SNS carries a per-message carrier fee — a few pence each in the UK — that varies by country and by how many texts you send, and it sits with your messaging spend, not your AWS compute bill. At 1,000 stops, if every customer got an SMS first notice and one update, that's a separate line in the tens of pounds, dwarfing the $3.10. That's exactly why the channel is a per-customer choice from Part 5: SMS for the stops where it genuinely earns its fee — tight windows, customers who must be present — and free email for everyone else.

What higher volume costs

Push this to a busy operator — 200 stops a day, five times the volume — and the AWS-side bill lands somewhere near $9, not $15.50. It's sub-linear because the fixed lines don't move: Secrets Manager stays at $0.40, EventBridge stays at a cent, AWS Budgets stays free. What scales is the genuinely usage-priced work — more geocode misses and bigger matrices on the routing API, more Bedrock messages, a bit more DynamoDB and logs — and even then the optimiser, the thing doing the actual clever work, stays close to free because it's plain Python on a function that runs for seconds.

The honest way to read this: the AWS bill is rounding error against the diesel. A round that doubles back on itself burns fuel and hours every single day; a few uncrossed legs and six timed stops hit on time pay for $3.10 many times over before lunch — and the customers stop ringing to ask where the van is.

DESIGN RULES THAT SHAPED THE COST

- Pay per morning, not per hour. No always-on compute means no idle bill.
- Cache the geocodes. Repeat streets are looked up once, which keeps the one real variable cost small.
- Buy the matrix once a day. The grid is computed per build, not per stop.
- Spend the model sparingly, or not at all. One Haiku call per message, swappable for a free template.
- Keep SMS off the AWS bill on purpose. Per-message carrier fees are a separate, opt-in line for the stops that need it.

PART 7 OF 7

JUNE 29, 2026 PART 7 OF 7 · DELIVERY ROUTE PLANNER SERIES ~8 MIN READ

Engineering reference: the delivery route planner architecture

This is the delivery route planner with the friendly labels removed: the real resource names, the runtime, the table key schemas, the morning schedule and the driver-event bus, the IAM scope, and where the routing service fits. If you want to build it rather than understand it, start here.

KEY TAKEAWAYS

- Eight Lambda functions, all Python 3.14 on arm64, with one SQS queue and a dead-letter queue between the build steps.
- Four DynamoDB tables, all on-demand: stops, routes, driver progress, and the geocode cache index.
- EventBridge Scheduler runs the morning build; a separate EventBridge bus carries the driver check-off events.
- Driver check-off arrives on a Lambda Function URL — no API Gateway; outbound is SES and SNS; the routing key is in Secrets Manager.
- One Bedrock model, Claude Haiku 4.5 via Global cross-Region inference, called only by the ETA function. Single region, `eu-west-2`.

The architecture, for engineers

This is the same system as Part 1 with the friendly labels removed and the real resources named. Everything is in one region, `eu-west-2` (London), in one account. There is no API Gateway, no NAT Gateway, and nothing always-on: the morning build is fired by EventBridge Scheduler, the one inbound HTTP surface (the driver check-off) is a Lambda Function URL, and the build steps are decoupled by a single SQS queue. The only thing outside AWS is the geocoding and distance-matrix provider.

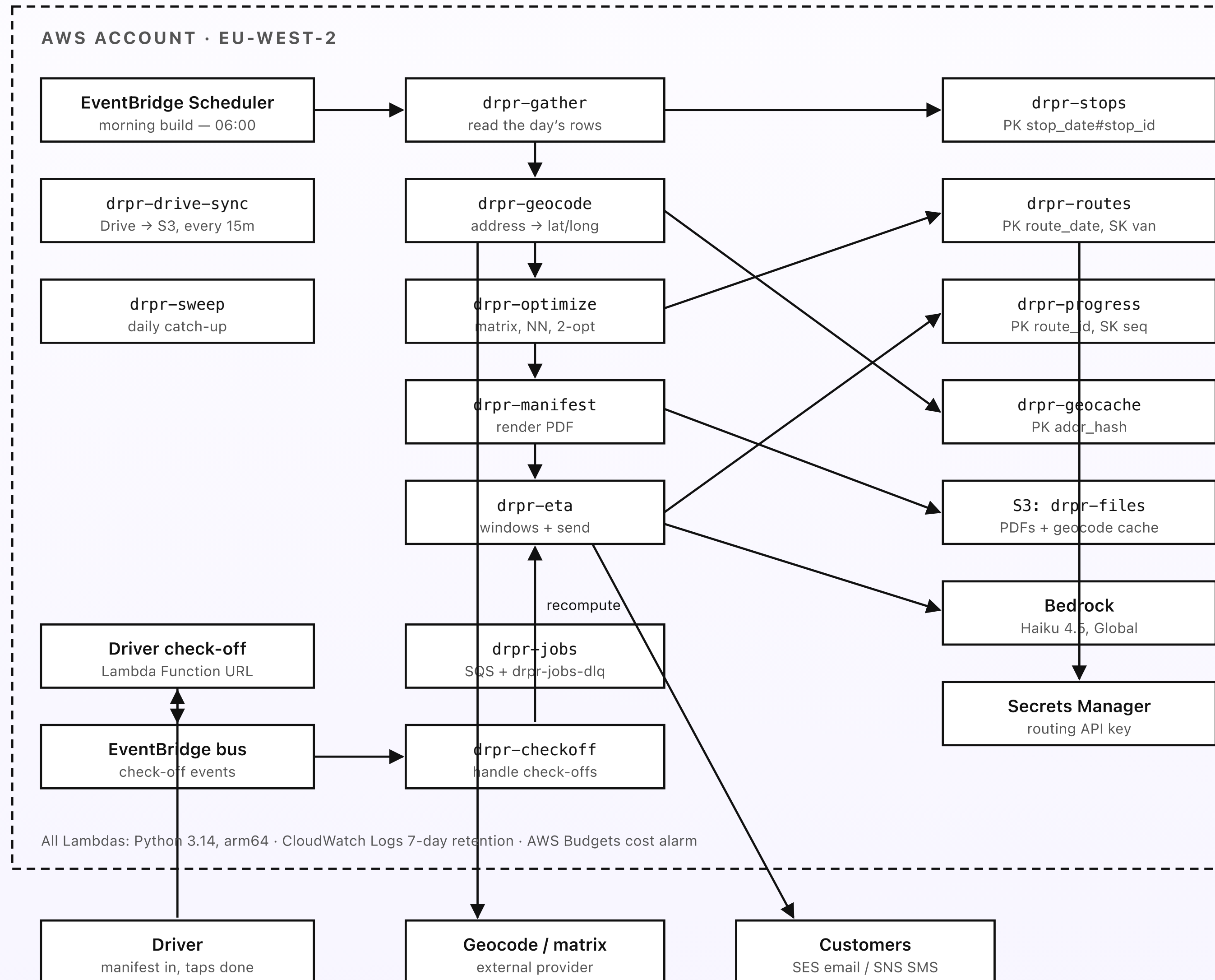

AWS ACCOUNT · EU-WEST-2
EventBridge Scheduler
morning build — 06:00
drpr-gather
read the day's rows
drpr-stops
PK stop_date#stop_id
drpr-drive-sync
Drive → S3, every 15m
drpr-geocode
address → lat/long
drpr-routes
PK route_date, SK van
drpr-sweep
daily catch-up
drpr-optimize
matrix, NN, 2-opt
drpr-progress
PK route_id, SK seq
drpr-manifest
render PDF
drpr-geocache
PK addr_hash
drpr-eta
windows + send
S3: drpr-files
PDFs + geocode cache
recompute
Bedrock
Haiku 4.5, Global
Driver check-off
Lambda Function URL
drpr-jobs
SQS + drpr-jobs-dlq
Secrets Manager
routing API key
EventBridge bus
check-off events
drpr-checkoff
handle check-offs
All Lambdas: Python 3.14, arm64 · CloudWatch Logs 7-day retention · AWS Budgets cost alarm
Driver
manifest in, taps done
Geocode / matrix
external provider
Customers
SES email / SNS SMS

Fig 7. The delivery route planner drawn for engineers: a scheduled build chain of five Lambdas, an SQS buffer, a Function-URL check-off feeding an EventBridge bus, four DynamoDB tables, S3, Bedrock called only by the ETA function, and one external routing provider. One region, one account, no API Gateway.

Lambda functions

Eight functions, all Python 3.14 on arm64, all with CloudWatch Logs at 7-day retention. Each does one job and hands off; the SQS queue (`drpr-jobs`, with `drpr-jobs-dlq` as its dead-letter queue after five attempts) decouples the build steps so a slow geocode or matrix call can't wedge the whole morning.

- `drpr-gather` — fired by EventBridge Scheduler. Reads the day's rows from the mirror, validates window, size, and address, writes clean rows to `drpr-stops` and flagged rows to a flagged list, and enqueues the build.
- `drpr-geocode` — resolves each address to a coordinate: reads `drpr-geocache` first, and only on a miss calls the external provider with the key from Secrets Manager, writing the result back to the cache.
- `drpr-optimize` — builds the distance/time matrix (hosted service or haversine fallback), runs the nearest-neighbour seed and 2-opt with time-window and capacity constraints, and writes the ordered route to `drpr-routes`.
- `drpr-manifest` — joins the route to the stop details, renders the manifest PDF, writes it to `drpr-files`, and sends the driver a link via SES or SNS.
- `drpr-eta` — computes a window per stop, makes the single Bedrock call for the wording, sends via SES/SNS, and on each check-off event recomputes the remaining windows from `drpr-progress`.

- `drpr-checkoff` — backs the driver check-off Function URL; validates the event, writes it to `drpr-progress`, and puts a check-off event on the EventBridge bus that wakes `drpr-eta`.
- `drpr-sweep` — scheduled daily. Catches rounds that never completed, stops with no check-off by end of day, and stale cache entries, and surfaces them to whoever runs the round.
- `drpr-drive-sync` — scheduled every 15 minutes. Mirrors the Drive stops sheet and settings doc into S3 and upserts into `drpr-stops` so the morning build never depends on a live Drive call.

Data stores, schedules, and messaging

- **DynamoDB (all on-demand).** `drpr-stops` — PK `stop_date#stop_id`, one clean stop per item with coordinate, window, size, and access note. `drpr-routes` — PK `route_date`, SK `van`, the ordered stop sequence and the matrix metadata for the day. `drpr-progress` — PK `route_id`, SK `seq`, the driver check-offs with actual completion times. `drpr-geocache` — PK `addr_hash` (the normalised-address key), the stored coordinate and confidence.
- **S3.** `drpr-files` — the rendered manifest PDFs (under a per-date prefix, lifecycle-expired after 30 days) and the geocode cache blobs that `drpr-geocache` indexes.
- **EventBridge.** Scheduler runs the morning build (a daily `cron` before the working day), the 15-minute `drpr-drive-sync`, and the daily `drpr-sweep`. A separate event bus carries the driver check-off events from `drpr-checkoff` to `drpr-eta`.

- **SES and SNS.** SES (verified domain, DKIM) sends the manifest link and the ETA emails; SNS sends the ETA SMS where a stop or the settings doc calls for it. Inbound customer replies route to a human, not back into the planner.
- **Secrets Manager.** One secret — the geocoding/matrix API 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 `drpr-eta`.

IAM scope and region

Each function gets its own execution role scoped to exactly what it touches, no wildcards. `drpr-gather` can read the mirror and write `drpr-stops`; it can't call Bedrock or the routing provider. `drpr-geocode` is the only role that can read the routing secret, and it can read and write `drpr-geocache` and nothing else. `drpr-optimize` reads `drpr-stops` and writes `drpr-routes` — no outbound network, no send permissions. `drpr-eta` is the only role with `bedrock:InvokeModel`, scoped to the one Haiku profile, plus SES and SNS send and read on `drpr-progress`; it cannot write the route. `drpr-checkoff` can write `drpr-progress` and put events on the bus, nothing more. Everything runs in `eu-west-2`; the only cross-Region path is Bedrock's Global inference profile, which routes the model call for capacity and holds no data. An AWS Budgets alarm watches the monthly spend and notifies if it drifts above a few dollars — the cheapest possible smoke detector for a runaway geocode loop or a stuck matrix call.

DESIGN RULES THAT SHAPED THE BUILD

- One job per function. Eight small Lambdas beat one that does everything; the queue decouples the slow calls.
- One public surface. Only the driver check-off Function URL is reachable from outside — no API Gateway.
- Least privilege, per role. Only `drpr-eta` calls Bedrock; only `drpr-geocode` reads the routing secret.
- State in DynamoDB, blobs in S3. Tables for stops, routes, progress, and the geocode index; S3 for PDFs and cache.
- One region, one model. `eu-west-2` throughout; Bedrock Haiku 4.5 via Global inference, called once per message.
- A budget alarm is a smoke detector. The cheapest way to learn a routing call looped is a Budgets alert.