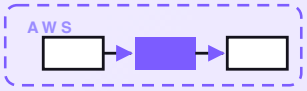
7-PART SERIES · FREE COMPANION



Shift scheduler

A serverless scheduler that takes the headache out of staff rotas for a small team. The manager sets who's available and the shifts that need covering; it drafts a fair weekly schedule that respects availability and hours, publishes it to staff once approved, and handles swap and time-off requests — finding a qualified replacement and routing the swap to the manager for a quick yes or no. It only proposes; the manager approves every schedule and every swap. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Shift scheduler

PART 1 OF 7

JUNE 16, 2026 PART 1 OF 7 · SHIFT SCHEDULER SERIES ~5 MIN READ

A shift scheduler on AWS for a few dollars a month

A small team's rota has more moving parts than it looks. Who's available which days. Who's trained to run the till and who isn't. Who's already done four shifts this week and shouldn't do a fifth. The person who asked for next Friday off three weeks ago and the manager who half-remembers saying yes. The Saturday opener who texts at 9pm asking if anyone can cover. Building the weekly rota by hand is a slow, thankless job, and it's the first thing that goes wrong when the manager is busy. This post walks through the design of a small scheduler that drafts a fair week, asks the manager to approve it, publishes it to staff, and handles swaps without anyone losing track.

KEY TAKEAWAYS

- Three sources for the week: a Drive sheet of staff and shifts, a time-off note lane, and a recurring template lane.
- Every shift ends in one of four outcomes on each draft: filled, fair-swap, short-staffed, or held.
- Fairness by rule: each person has a weekly hours target, and the draft prefers whoever is furthest below it.
- The scheduler only proposes. The manager approves every draft and every swap before anything reaches staff.
- Designed on AWS for about $2.20/month at typical small-team volume.

The whole system on one page

Before any code, here’s the shape of what we’re designing.

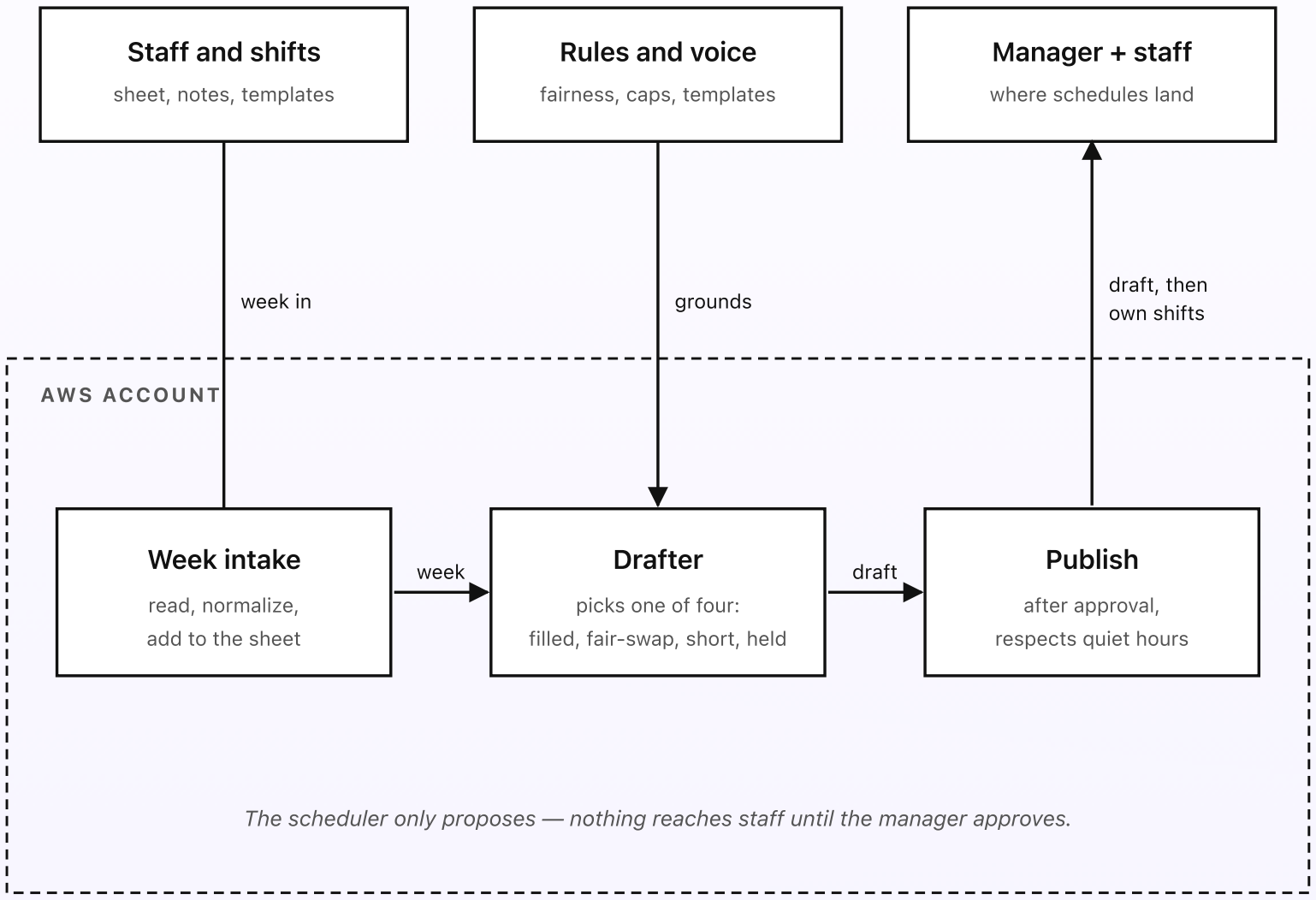

Staff and shifts
sheet, notes, templates
Rules and voice
fairness, caps, templates
Manager + staff
where schedules land
week in
grounds
draft, then
own shifts
AWS ACCOUNT
Week intake
read, normalize,
add to the sheet
week
Drafter
picks one of four:
filled, fair-swap, short, held
draft
Publish
after approval,
respects quiet hours
The scheduler only proposes — nothing reaches staff until the manager approves.

Fig 1. Three sources outside, three pieces inside AWS. The week flows in from a Drive sheet, a time-off note lane, and a recurring template lane. The Drafter runs weekly and picks one of four outcomes per shift. Publish sends each person their own shifts once the manager approves.

What you set up once (the outside)

- **Staff and shifts.** A Google Sheet in a Drive folder. One tab lists staff: name, role, weekly hours target, the skills they're cleared for (open, close, till, kitchen, first-aid), a Slack ID or email, and a max-hours cap. Another tab holds each week: the shifts that need covering (day, start, end, role, skills needed) and who's available. You can fill it in once and reuse it; new things can also enter via two other lanes covered in Part 2 — a time-off note lane (a staffer emails "off next Friday for a wedding" and the scheduler reads it into dates for one-tap approval) and a recurring template lane (the shifts that look the same every week get filled in for you).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers the fairness settings — the max-hours cap per person, the minimum rest gap between shifts (so nobody closes at 11pm and opens at 7am), the skill requirements per shift, and how strongly to balance hours against everyone's target. The *voice* doc holds the message templates — what the schedule DM and the swap request actually say.
- **Manager and staff.** The manager approves every draft and every swap. Each staffer has a Slack member ID (so their shifts arrive as a private DM) or, if Slack isn't set up for them, an email address. Once the manager approves, each person gets only their own shifts — the day, the time, the role — plus a button to request a swap or flag a problem.

What runs on every draft (the inside)

- **The week intake.** Three sources feed the week. The Drive sheet is the canonical store. New things can also be added via the time-off note lane (a staffer emails `timeoff@your-company.com`, the scheduler uses Bedrock Haiku 4.5 to read the plain-English note into a date range, then drops a one-tap approval card in the manager's Slack to confirm before it's marked on the week) and the recurring template lane (a small sync Lambda copies the standing weekly pattern into next week's tab so the manager only edits the exceptions).
- **The drafter.** Runs once a week, by default Thursday at 2pm local, in time for next week. Reads the availability, the hours targets, and the skills. For each shift, it places a qualified, available person, preferring whoever is furthest below their weekly hours target. Picks one of four outcomes. *Filled*: a clear best person was available and placed. *Fair-swap*: two people both fit, so the one further below target gets it to even out the hours. *Short-staffed*: a shift has fewer qualified, available people than it needs — flag it for the manager. *Held*: a shift the manager marked to assign by hand. The drafter itself doesn't call a model — the placement logic is plain Python.
- **Publish.** The full draft goes to the manager first, with an Approve, Edit, or Re-draft button and the fairness summary attached. Nothing goes to staff until the manager taps Approve. On approval, each person gets only their own shifts as a Slack DM (or email), plus a calendar invite per shift. Both honor quiet hours (no pings between 9pm and 7am local by default). Swap requests come back the same way — a qualified replacement is found and the swap is routed to the manager for a quick yes or no. Every send and every approval writes a row in DynamoDB. A weekly fairness summary shows each person's hours against target.

In plain words

It's Thursday afternoon. The drafter reads next week: eleven staff, sixty-two shifts to cover, three people who asked for days off, two who can run the kitchen and four who can't. It places everyone by rule, keeping hours close to each person's target, and lands a clean draft except for one short-staffed Saturday night that needs a kitchen-trained person who's already at their cap. That one shift gets flagged. The whole draft — plus the flag — lands in the manager Dana's Slack at 2:03pm with an Approve button and a one-line note: "Sat night kitchen is short — only Priya is cleared and she's at 38h." Dana taps Edit, moves one shift, asks Priya directly about the Saturday, then taps Approve. Each person gets their own shifts and a calendar invite. On Friday, Marco texts that he can't do Sunday; he taps Request swap, the scheduler finds two cleared people who are under their hours and free, and routes the best one to Dana for a yes. Dana taps yes. Marco's Sunday is covered, and nobody had to run a group chat.

The cost of running this is about $2.20 a month at small-team volume. The cost of *not* running it is the Saturday nobody was scheduled to open, the person quietly given every weekend until they quit, and the hour every week the manager spends rebuilding the same grid by hand.

DESIGN RULES THAT SHAPED EVERY DECISION

- The scheduler only proposes. The manager approves every schedule and every swap — nothing reaches staff on its own.
- Four outcomes, always. Filled, fair-swap, short-staffed, held. There is no fifth.
- Fairness is a rule, not a vibe. Each person has an hours target and the draft balances against it.
- Quiet hours are respected. A shift ping at 11pm is worse than one at 8am the next day.
- The staff list lives in Drive. Adding a person, changing a skill, or shifting hours doesn't need a deploy.
- Every draft, approval, and swap is logged. Check why a shift went to someone next month and it's all there.

Why this shape

Most small teams build the rota one of three ways: a spreadsheet the manager rebuilds every week, a group chat where shifts get claimed and lost, or one person's memory. The spreadsheet works until the manager is on holiday and nobody else knows the rules. The group chat is chaos with a paper trail nobody reads — two people think they swapped, neither is sure. And memory fails the moment the person who held it isn't there, which is exactly when the rota matters most.

The setup above keeps the staff list in a sheet the team already edits, but adds a small system that *builds* the week from it and acts only when something needs a human. The draft comes early enough to fix. It balances hours so the same person isn't quietly carrying the team. It flags the shifts it can't fill instead of pretending they're fine. And it never publishes or confirms a swap on its own — the manager stays in control of who works when. The scheduler is invisible most of the week; it shows up on Thursday afternoon and whenever somebody needs to swap.

The next four posts walk through each piece in turn: how a shift week gets set up, how a fair rota gets drafted, how a schedule reaches the team, and how a shift swap gets approved. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 16, 2026 PART 2 OF 7 · SHIFT SCHEDULER SERIES ~4 MIN READ

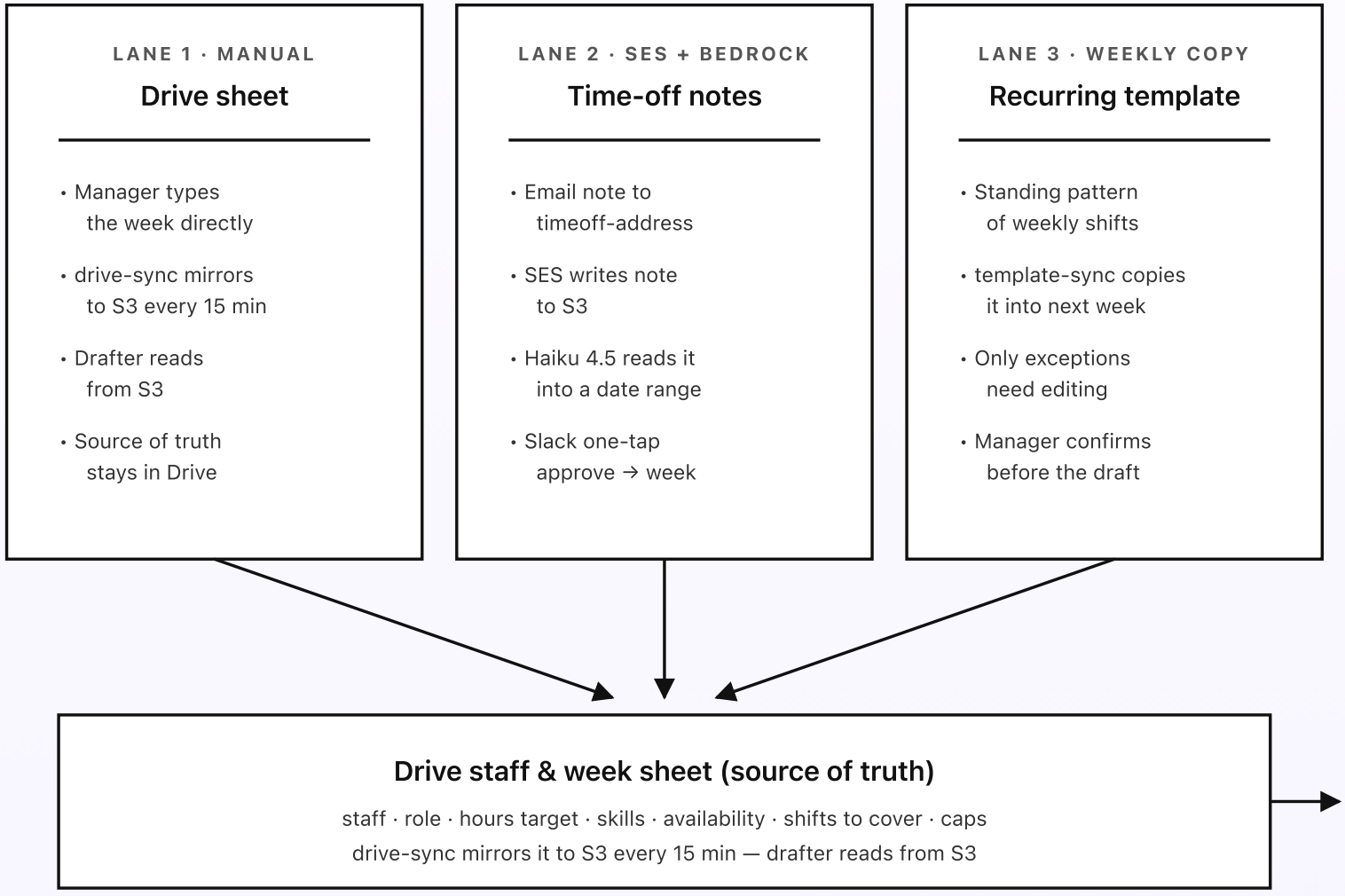
How a shift week gets set up

The drafter can only build from what's in the sheet. So the first job is making sure each week's sheet reflects who's actually free and what actually needs covering. There are three ways things get in: somebody types into the Drive sheet, somebody emails a plain-English time-off note, or the standing weekly pattern gets copied in for you. The first one is obvious. The other two exist because in real life nobody updates a sheet for the day off they asked about in passing, and nobody wants to retype the same Monday-opener shift fifty-two times a year.

KEY TAKEAWAYS

- Three intake lanes feed one sheet: the Drive sheet, a time-off note lane, and a recurring template lane.
- Plain-English time-off notes are read by Bedrock Haiku 4.5 into a clean date range.
- Every read-in note goes to the manager's Slack for one-tap approval before it lands on the week.
- The recurring template lane copies the standing weekly pattern so only exceptions need editing.
- The sheet stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one sheet



The Drive sheet stays the source of truth — the other lanes are conveniences that propose changes for it.

Fig 2. Three lanes converge on one Drive sheet. The sheet is the source of truth; the time-off lane and the template lane are conveniences that propose changes for manager approval. The drive-sync Lambda mirrors the sheet to S3 so the drafter can read it without hitting Drive on every run.

Lane 1: the Drive sheet itself

The simplest lane. Open the sheet in Drive, edit the week, save. The staff tab is short: name, role, weekly hours target, skills cleared, Slack ID or email, and a max-hours cap. The week tab holds the shifts that need covering (day, start, end, role, skills needed) and each person's availability. A small Lambda — `drive-sync` — runs every fifteen minutes, exports the sheet as plain CSV via the Drive API, and writes it to `s3://ss-roster-source/roster.csv` if the sheet has changed since the last sync. The drafter reads from S3, not Drive directly. That keeps Drive API calls predictable and gives you S3 versioning for free, so a bad bulk-edit can be rolled back in one click.

This lane covers the cases where the manager already knows the week — the new hire's first shift, the festival weekend that needs extra cover, the person who's out all week. Most of the week goes in this way.

Lane 2: time-off notes (the lane most teams actually use)

Set up a dedicated inbound address — something like `timeoff@your-company.com` — via Amazon SES. Anyone on the team emails it in plain English and the scheduler takes it from there. SES writes the raw message to `s3://ss-raw-mime/`. The S3 PUT triggers a parser Lambda. The Lambda pulls out the note text and calls Bedrock Haiku 4.5 to read it.

The note “off next Friday for a wedding, back Monday” becomes a structured result: who sent it (from the email address), a start date, an end date, and a short reason. The model prompt is plain: “Read this time-off note. Return JSON only with start date, end date, and reason. Use today’s date to resolve words like ‘next Friday.’ Do not invent a date that isn’t implied by the note.” The result goes to a small Slack message that pings the manager: the person, the dates read, the reason, and three buttons — *approve*, *edit*, *decline*. On *approve*, a Lambda marks those days as unavailable for that person via the Sheets API. On *edit*, the manager gets a fillable modal pre-filled with the read-in dates. On *decline*, the request is logged and the staffer gets a short note back.

The reason every read-in note goes to the manager first is simple: a time-off request the model misread is worse than one that never made it into the sheet. A wrong date quietly tells the drafter someone’s free when they’re not, and the short-staffed shift shows up the morning of, not the week before.

Lane 3: recurring template

Most weeks look a lot like the last one. The same Monday opener, the same Friday-night double, the same two cleaners on Wednesday. Forcing the manager to retype that grid every week is busywork that invites mistakes.

Lane 3 keeps a standing weekly pattern in its own tab — the shifts that repeat. A small `template-sync` Lambda runs once a week, copies the pattern into next week’s tab, and leaves the manager to edit only the exceptions: the holiday, the extra cover, the person who’s away. Anything the manager hasn’t touched is assumed unchanged. If the template would clash with an approved time-off note,

the conflict is surfaced in the same Slack flow as Lane 2 so it's caught before the draft runs.

The template lane is the most opt-in of the three. A team with a wildly different week every week can ignore it; a team with a steady rhythm avoids rebuilding the same grid fifty-two times a year.

Why the sheet stays the source of truth

Three lanes in, but only one place where the drafter actually looks. That's a deliberate constraint. If two lanes both wrote directly to the drafter's state, every "why was this person scheduled?" question would mean checking three places. Funneling everything through the Drive sheet means there is exactly one view of the week, and the manager can read or edit any of it without learning a new tool. The convenience lanes are first-class for getting things in, but they always pass through the sheet on the way.

Next post: how the drafter actually reads the week, balances hours, and picks one of four outcomes per shift.

PART 3 OF 7

JUNE 16, 2026 PART 3 OF 7 · SHIFT SCHEDULER SERIES ~5 MIN READ

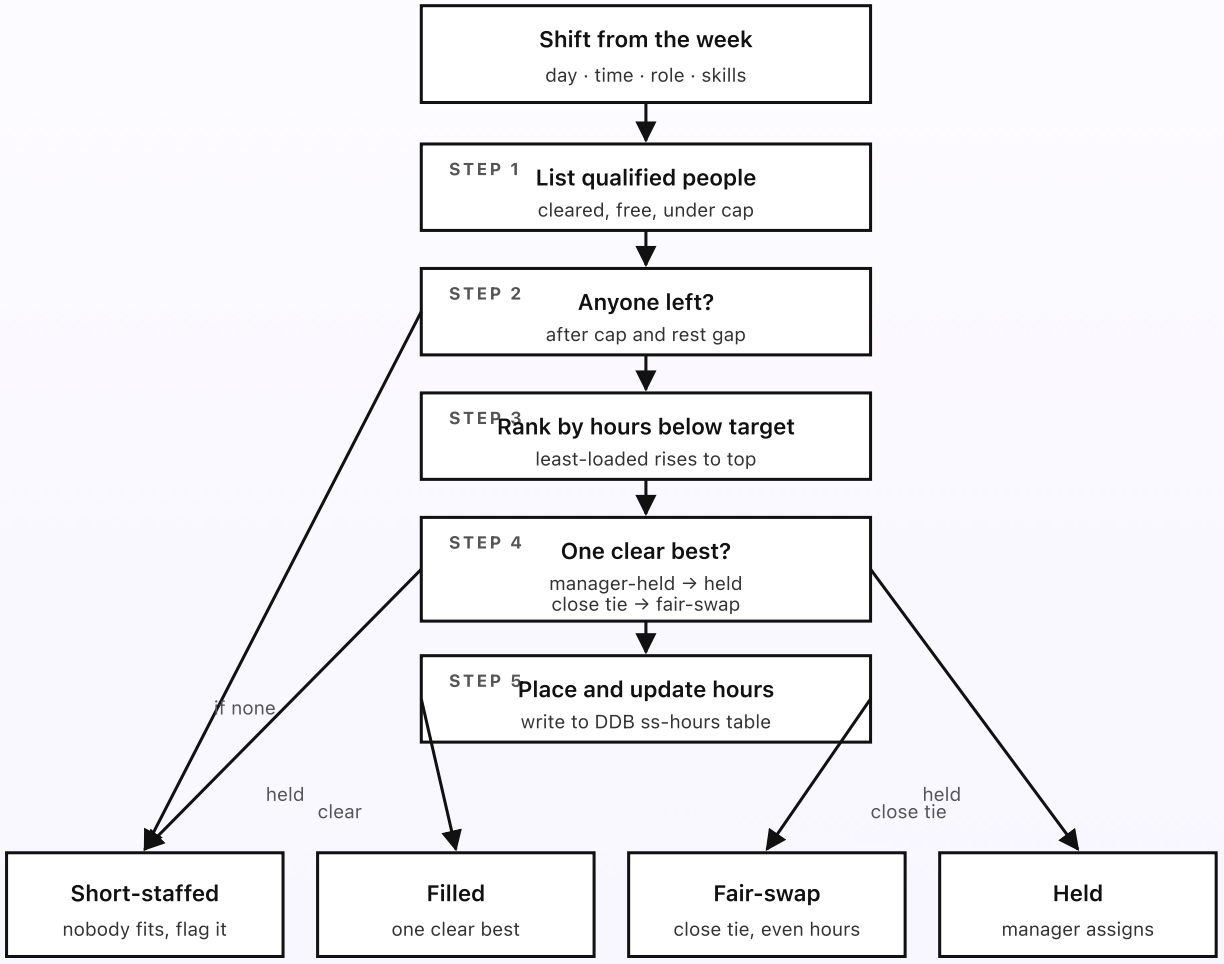
How a fair rota gets drafted

Once a week, by default Thursday at 2pm local time, an EventBridge Scheduler rule fires the drafter Lambda. The Lambda reads the week, sorts the shifts, and fills them one at a time, deciding for each shift who's the fair, qualified, available pick — or flagging it if nobody fits. The whole decision is plain Python. No model. No vector search. Every setting lives in the rules doc, where the manager can edit it without a deploy.

KEY TAKEAWAYS

- The drafter runs once a week via EventBridge Scheduler, by default Thursday at 2pm local time.
- Settings live in the rules doc — max-hours caps, rest gaps, skill requirements, and how hard to balance hours.
- Four outcomes per shift, every run: filled, fair-swap, short-staffed, held.
- DynamoDB tracks a running hours total per person so the draft balances load as it goes.
- The drafter itself never calls a model. The placement is entirely deterministic.

The decision flow, per shift



The rules doc holds every setting — change a cap and the next run uses the new value.

Fig 3. The drafter's decision tree, per shift, per weekly run. Five steps decide which of four outcomes applies. The rules doc holds every setting; the drafter only enforces them.

Fairness: the hours target isn't magic, it's in the sheet

Each person has a weekly hours target in the staff sheet — the full-timer is set to 38, the student to 12, the weekend-only person to 16. As the drafter fills shifts, it keeps a running total of the hours it's already placed on each person this week. For every open shift, it prefers whoever is furthest below their target. Over a week, that evens the load: nobody ends up with every closing shift while someone else gets nine hours, unless availability genuinely forces it.

The settings exist for a reason. The max-hours cap stops the draft from quietly handing one person a sixth shift. The rest gap (default eleven hours) stops the close-then-open double that burns people out. The skill requirements stop a kitchen shift going to someone who's never run the kitchen. Different shifts have different needs; the rules reflect that.

Per-person overrides exist too. The staff sheet has an optional column called `cap_override`. Set a number there and the drafter uses it instead of the default cap for that one person. This is the right escape hatch for the manager who agreed to let a senior staffer pick up extra hours over the holidays.

Four outcomes, always

Every shift, every run, lands in exactly one of four buckets. The names are simple on purpose.

- **Filled.** One qualified, available person was clearly the fair pick — furthest below their target — and got placed. Their running hours total in the `ss-hours` DynamoDB table is updated. Most shifts, most weeks, land here.
- **Fair-swap.** Two people both fit and both sit close on hours. The shift goes to the one further below target to even things out, and a short note records why. This is the rule that keeps the rota feeling fair instead of arbitrary.
- **Short-staffed.** A shift has fewer qualified, available people than it needs — everyone cleared is either off, at their cap, or inside a rest gap. The drafter doesn't pretend it's fine. It flags the shift for the manager with the reason ("only Priya is cleared and she's at her cap") so a human can decide.
- **Held.** A shift the manager marked to assign by hand — the trial shift for the new hire, the sensitive Saturday the manager wants to place personally. The drafter leaves it open and notes it as held. The manager fills it during the approval step in Part 4.

State that makes the draft deterministic

The drafter reads and writes one DynamoDB table as it goes. `ss-hours` holds the running placed-hours total per person for the week being drafted: `(person_id, week_id, hours_placed)`. With that table plus the week from the sheet and the settings from the rules doc, the placement logic is a few dozen lines of Python and zero magic. The same week, the same availability, the same settings always produce the same draft. Re-running the draft for a week produces the same rota

(the run clears and rebuilds `ss-hours` for that week each time, so there's no double-counting).

Approving a draft freezes it: the placements become the published schedule, and the `ss-hours` totals become the baseline that swaps in Part 5 adjust against. Re-drafting before approval just rebuilds from scratch.

Why the weekly draft uses no model

The drafter could call a model to "figure out a good rota." It doesn't. Two reasons. First, the draft should be the one part of the system the manager can fully reason about — if the rules say cap at five shifts and respect an eleven-hour rest gap, that's exactly what happens, every time. A model in that loop would make "why did Sam get this shift?" impossible to answer cleanly. Second, model calls cost money and add latency, and a rule-based placement is both cheaper and more explainable for a problem this well-defined.

Bedrock fires elsewhere — on the time-off note lane in Part 2, and on the weekly fairness summary mentioned in Part 6. Not on the draft. The drafter itself is plain Python that reads a sheet and writes placements.

Next post: how an approved schedule reaches the team, how quiet hours are honored, and what each person actually sees.

PART 4 OF 7

JUNE 16, 2026 PART 4 OF 7 · SHIFT SCHEDULER SERIES ~5 MIN READ

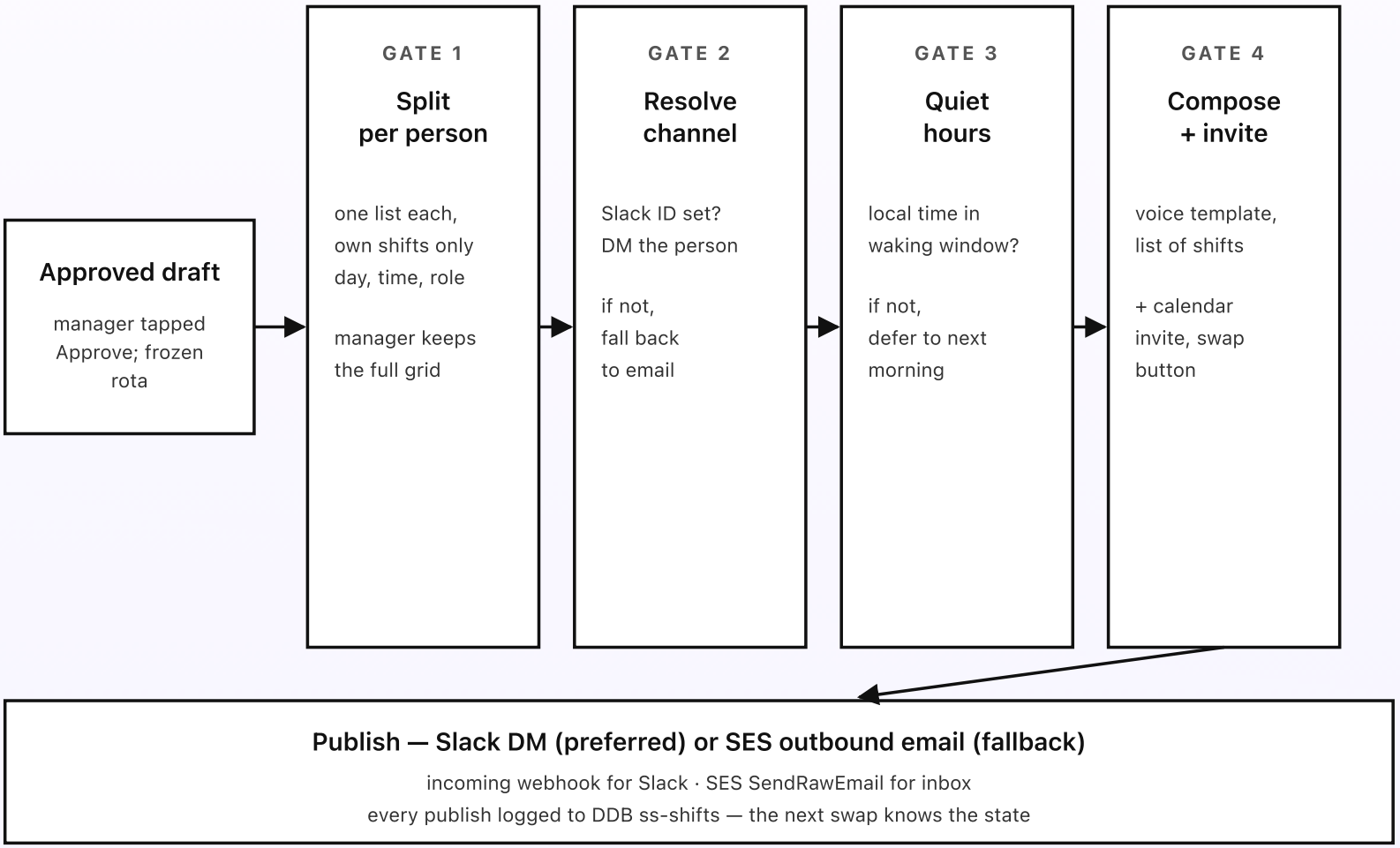
How a schedule reaches the team

The drafter built a week and the manager tapped Approve. Now the publish Lambda has to get each person their own shifts, on the right channel, at a reasonable time of day, with enough detail to act on. Get any of those wrong and the schedule is worse than a printed sheet on the wall: a 1am Slack ping, the whole team's rota dumped on everyone, an invite to someone who left last month. And the schedule never reaches anyone at all until the manager approves it first. Four small guardrails sit between the approved draft and the schedule landing.

KEY TAKEAWAYS

- Nothing publishes until the manager taps Approve on the full draft — Edit and Re-draft are the other two options.
- Each person gets only their own shifts; the whole-team grid stays with the manager.
- Quiet hours defer pings to the next reasonable hour; calendar invites go out per shift.
- Slack DMs are the default; email is the fallback if no Slack ID is configured.
- Every publish is logged so the next swap or re-draft knows the current state.

Four guardrails on every publish



Every gate is a deterministic check — no model calls, no surprise ping at 1am on a Sunday.

Fig 4. Four guardrails between the approved draft and the published schedule. Split per person. Resolve the channel. Honor quiet hours. Compose with calendar invites. Then ship via Slack or email and log the publish so the next swap starts from the right state.

Approval comes first

Before any of the gates run, the full draft goes to the manager and nobody else. It arrives as a Slack message (or email) with the whole-team grid, the fairness summary from Part 3, any short-staffed flags, and three buttons: *Approve*, *Edit*, *Re-draft*. *Approve* freezes the rota and starts the publish. *Edit* opens the sheet so the manager can move a shift or fill a held one, then approve. *Re-draft* sends it back to the drafter — useful after the manager changes availability or a setting. Nothing reaches a single staff member until Approve is tapped. This is the heart of the system: the scheduler proposes, the manager decides.

Gate 1: split per person

The whole-team grid is the manager's view, not the team's. Gate 1 turns the approved rota into one private list per person, containing only that person's shifts: the day, the start and end time, and the role. Nobody on the floor sees the full rota, which keeps the schedule from turning into a running comparison of who got what. The manager keeps the full grid; everyone else gets just their own week.

Gate 2: resolve the channel

For each person, the publish Lambda looks up their delivery preference from the staff sheet. If a Slack member ID is set, the schedule goes as a private Slack DM — it feels like a work message and carries action buttons. If no Slack ID is set, it falls back to the person's email address. Email is the fallback so the new hire who isn't on Slack yet still gets their shifts. Nobody falls through the cracks.

Gate 3: quiet hours

The draft usually runs Thursday afternoon, so an approval that follows right away is already in waking hours. But managers approve when they get a minute — sometimes that's late at night. Gate 3 reads the rules doc's quiet-hours setting (default 9pm to 7am, configurable per business). If the current local time is in the quiet window, the publish creates a one-off EventBridge Scheduler rule that fires at the next reasonable minute and exits without sending. The Scheduler re-invokes the publish Lambda with the same payload in the morning, where Gate 3 lets it through. A schedule approved at 11pm lands at 7am, not at 11pm.

Gate 4: compose, invite, then ship

The voice doc has the schedule message template: a short greeting, the person's shifts for the week as a clean list, and a line about how to request a swap. The publish Lambda fills it in, attaches a calendar invite per shift (an `.ics` file or a Google Calendar event, depending on setup), and adds a "Request swap" button. For Slack, the message ships via the incoming webhook; the webhook URL lives in Secrets Manager. For email fallback, the same content is wrapped in a small HTML

email, and the “Request swap” link hits a Function URL — the email equivalent of the Slack button.

Every publish — Slack or email — writes a row to `ss-shifts` in DynamoDB: who’s working what, this week, as published. When a swap comes in (Part 5) or the manager re-drafts, that table is the current truth the change is applied against.

Why the guardrails exist

None of these gates are exotic. They’re the kind of small care a thoughtful manager would take handing out the rota themselves — give each person just their own shifts, send it on a channel they actually check, don’t text at midnight, include the times and a way to flag a problem. Putting them in code as four small sequential gates, behind a manager approval that gates everything, makes the care part of the design rather than something you’re trusting a busy week to remember.

Next post: how a shift swap gets approved once the schedule is out — how a qualified replacement is found, how the swap is routed to the manager, and how the hours stay fair.

PART 5 OF 7

JUNE 16, 2026 PART 5 OF 7 · SHIFT SCHEDULER SERIES ~5 MIN READ

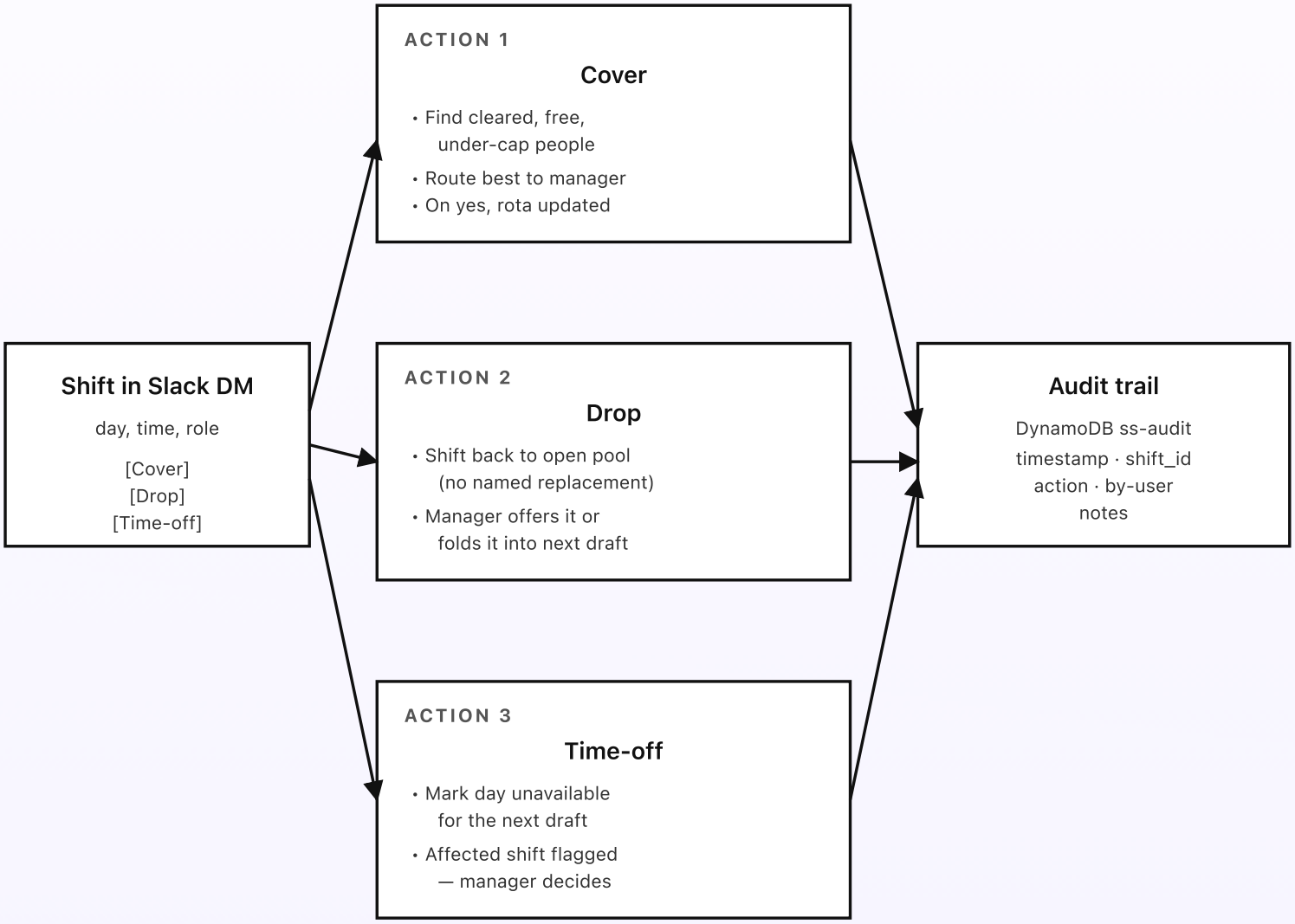
How a shift swap gets approved

A schedule lands in Marco's Slack DM on Friday morning. He's got Sunday 4–10pm, and something just came up. There's a "Request swap" button. What happens when he taps it? The honest answer is "it depends on what he's actually asking for." This post walks through the three things a swap request can be — cover, drop, time-off — and how the published rota, the hours, and the audit trail all stay in sync while the manager keeps the final say.

KEY TAKEAWAYS

- Three actions per request: *cover* (find a replacement, route to manager), *drop* (back to the open pool), *time-off* (hold for next draft).
- Cover reuses the Part 3 rules — only cleared, available, under-cap people are offered.
- Every swap routes to the manager for a quick yes or no. Nothing is auto-confirmed.
- An approved swap updates the published rota and the running hours so fairness stays intact.
- The Request-swap button is a Slack interactive message backed by a Function URL.

Three actions on a swap request



Every swap needs a manager yes — the scheduler proposes the replacement, never reassigns on its own.

Fig 5. Three actions per swap request, three different effects. Cover finds a replacement and routes it to the manager. Drop returns the shift to the open pool. Time-off holds the day for the next draft. Every action needs a manager yes and writes to the audit trail.

Action 1: cover (the most common)

Marco can't do Sunday but wants it covered so the floor isn't short. He taps *Cover*. The Function URL Lambda reuses the exact rules from Part 3: it lists everyone cleared for the shift's role and skills, free that day, and under their max-hours cap and rest gap. It ranks them by who's furthest below their weekly hours target — the same fairness logic the draft uses — and picks the best one.

That proposed swap goes to the manager as a Slack message: "Marco wants Sunday 4–10pm covered. Best fit is Aisha (cleared, free, at 14h of 20h). Approve?" with a yes or no. On *yes*, three things happen, in order. First, the published rota in `ss-shifts` is updated: the shift moves from Marco to Aisha. Second, both people's running hours in `ss-hours` are adjusted — Marco down, Aisha up — so fairness stays honest. Third, a `action: swapped` row is written to `ss-audit` with both names, the shift, and the timestamp. Aisha and Marco each get a short confirmation DM. On *no*, Marco is told the swap wasn't approved and the shift stays his.

The manager is never handed a blank "someone wants to swap, sort it out" message. They get a specific, already-fair proposal and a single decision to make.

Action 2: drop (back to the pool)

Sometimes a staffer can't cover the shift themselves and doesn't know who can — they just need it off their plate. *Drop* is for that. The shift is marked open in `ss-shifts` rather than reassigned to anyone, and it's surfaced to the manager as an open shift that needs a home.

From there the manager has choices: offer it to the team (the scheduler can post it to a shared channel as "open shift — Sunday 4–10pm, who can take it?" and the first qualified taker routes back for a yes), or simply fold it into the next draft so the drafter places it fresh. Either way, nothing is reassigned without the manager. Drop is the honest version of "I can't, can someone else?" — it leaves a clear open shift instead of a vague group-chat plea.

Action 3: time-off (the day-off ask)

Sometimes the request isn't really about one shift — the person needs the whole day, or a stretch of days, off. *Time-off* handles that. It records the unavailability against the week so the next draft treats the person as off, exactly like an approved Lane 2 note from Part 2. The affected shift on the current published rota is flagged for the manager, who decides whether to cover it now (via the same cover flow) or let the next draft sort it.

Time-off doesn't silently rewrite the published week. It marks the intent and surfaces the gap, so the manager can choose between covering it immediately and waiting for the redraw. If the day was already published as a shift, that shift stays Marco's until the manager either approves a cover or re-drafts — the system never leaves a shift quietly uncovered.

Every action is logged, every action is reversible

The `ss-audit` table records every cover, drop, and time-off with the person who asked, the manager who decided, the timestamp, and a snapshot of the rota before and after. If a swap gets approved by mistake (wrong person, wrong shift), the manager can run an “undo last swap” through a small admin command that reads the previous-state snapshot and restores both the rota and the hours. The undo is itself an audit row, so the trail of changes stays clean.

This kind of reversibility matters because swaps happen fast and under pressure — the Friday-night “can anyone cover” is exactly when mistakes creep in. The audit trail is the memory the manager can lean on when someone asks, three weeks later, why they ended up with that Sunday.

Next post: the cost breakdown. The whole system above runs in coffee-money territory at small-team volume; Part 6 explains exactly where the dollars go and why it stays cheap.

PART 6 OF 7

JUNE 16, 2026 PART 6 OF 7 · SHIFT SCHEDULER SERIES ~3 MIN READ

What the shift scheduler costs

The scheduler is one of the cheapest systems in this whole series. The weekly draft reads a CSV from S3, does some matching, writes a few rows to DynamoDB, and posts a handful of messages to Slack. It calls no models on the draft. Bedrock fires only when somebody emails a plain-English time-off note and once a week for the fairness summary. At typical small-team volume, the bill is a couple of dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $2.20/month at typical small-team volume (around 15 staff, 80 shifts a week).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The weekly draft costs pennies — no model calls.
- Bedrock fires only on time-off notes (a few a week) and the weekly fairness summary.
- At 30 staff the bill is around $5. At 60 staff it's around $9.

Cost at three volumes

The weekly draft and publish are the dominant cost — and even those are fractions of a cent per person.

Fig 6. Monthly cost at three team sizes. Bedrock and SES are small slivers because they only fire on the time-off note lane and the weekly summary. The dominant cost is the everything-else bucket: the weekly draft and the per-person publish.

Where the dollars actually go

Lambda runtime (the bulk). The drafter runs once a week. Each run reads the roster CSV from S3, sorts the shifts, and matches people into them. At 15 staff and 80 shifts, that's a few hundred milliseconds. At 60 staff it's a couple of seconds. Either way it's pennies a month. Add the publish Lambda firing per person each week, the Function URL Lambda for approvals and swaps, the calendar invites, the time-off parser, and the drive-sync Lambda every fifteen minutes — the Lambda total still lands around a dollar at all three sizes.

DynamoDB on-demand. Three small tables: `ss-shifts`, `ss-hours`, `ss-audit`. Writes happen on each draft, publish, and swap. Reads dominate during the weekly run. Pennies a month at any of these sizes.

S3 + Storage. The mirrored roster CSV plus the archived MIME from any forwarded time-off notes. A few hundred KB total at small-team volume. Effectively free.

EventBridge Scheduler. The weekly draft rule, the drive-sync and template-sync rules, plus deferred publish rules from the quiet-hours gate. A handful of invocations a week. Pennies.

SES. Inbound for the time-off note lane: $0.10 per thousand received messages (so a couple of cents a year for a small team). Outbound for email-fallback schedules: $0.10 per thousand sent. Both are negligible at this scale.

Bedrock (only when something fires it). The weekly draft uses no Bedrock. The time-off note lane fires Haiku 4.5 once per note: a few hundred input tokens (the note text) and a few dozen output tokens (the date range JSON), so a fraction of a

cent per note. At a few notes a week, Bedrock costs cents. The weekly fairness summary is one slightly larger call: write a short paragraph on each person's hours against target; a fraction of a cent.

SES inbound parsing. The time-off notes are plain text, so there's no Textract or document parsing here — just the small SES receive charge and the Bedrock read. That keeps this lane cheaper than the document-heavy lanes in some of the other systems in this series.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the approve and swap endpoints.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The drafter wakes once a week and sleeps the rest.
- **A Knowledge Base.** The roster is structured rows, not free text — rule-based matching beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models on the draft.** The weekly placement is plain Python. Bedrock fires only on time-off notes and the weekly summary.

How the cost scales

Lambda runtime grows roughly linearly with team size, because the draft matches every shift and the publish sends to every person. DynamoDB grows linearly too.

Bedrock and SES are uncorrelated with team size — they only fire when somebody sends a time-off note or it's the weekly summary. So the bill at 120 staff is around $18; at 250 it's around $38. Past those sizes a single weekly draft probably stops being the right shape (you'd split the roster by location or department and draft each separately), but those are scaling choices for a much bigger operation — not redesigns.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. The small-team bill stays well under that ceiling.

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

PART 7 OF 7

JUNE 16, 2026 PART 7 OF 7 · SHIFT SCHEDULER SERIES ~8 MIN READ

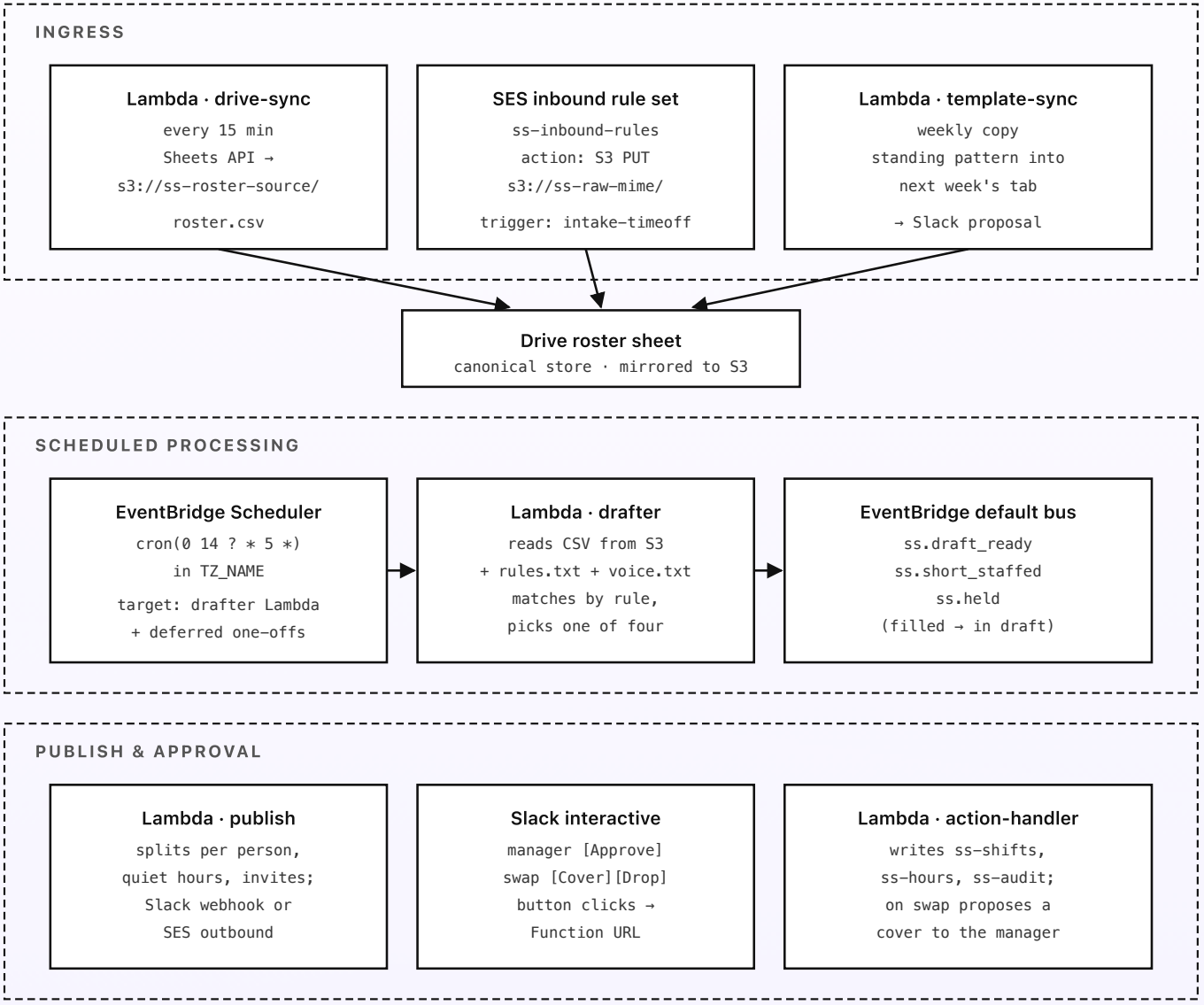
Engineering reference: the shift scheduler architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES inbound rule set, EventBridge Scheduler config, the DynamoDB schemas, and the Slack interactive flow. Read alongside the previous six posts; this one's the build sheet.

Region and account shape

Default region: **ap-southeast-1** (Singapore). SES inbound, Bedrock cross-Region inference, and EventBridge Scheduler are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at small-team volume — the failure mode for a small team is a manager publishing a rota a few hours late, not a regional outage. One AWS account dedicated to the scheduler (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



The scheduler only proposes — and every interaction is logged to ss-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the roster), scheduled processing (the weekly drafter emitting events), publish and approval (the schedule ships after the manager approves and the team's responses are recorded). Every Lambda is event- or schedule-driven; nothing is synchronous-chained.

Lambda functions

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

- **`drive-sync`** — EventBridge Scheduler target, fires every 15 minutes. Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `ss/drive/sa`) to export the roster sheet as CSV and write to `s3://ss-roster-source/roster.csv` only if the sheet has changed since the last sync. Same pattern syncs the rules and voice docs to `s3://ss-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`template-sync`** — EventBridge Scheduler target, weekly (a few hours before the draft). Copies the standing weekly pattern tab into next week's tab via the Sheets API, then checks the result against approved time-off; any clash becomes a Slack interactive proposal for the manager. Memory: 256 MB. Timeout: 30 s.
- **`intake-timeoff`** — S3 PUT trigger on `s3://ss-raw-mime/`. Parses MIME, extracts the note text, and calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-`

`v1:0`) to read the plain-English request into `{start_date, end_date, reason}`, resolving relative dates against the configured timezone. Posts the proposal to the manager's Slack with Approve/Edit/Decline buttons. No Textract — the notes are plain text, not documents. Memory: 512 MB. Timeout: 30 s.

- **`drafter`** — EventBridge Scheduler target, weekly on Thursday at 2pm local time (the schedule expression runs in `TZ_NAME` set to the team's timezone, e.g. `Asia/Singapore`). Reads `s3://ss-roster-source/roster.csv` and the rules and voice docs. Sorts shifts, lists qualified-and-available candidates per shift, ranks by hours-below-target, places each, and tracks running hours in `ss-hours`. Emits the assembled draft plus one event per flagged shift: `ss.short_staffed` or `ss.held`; the whole draft goes to the manager as `ss.draft_ready`. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls.*
- **`publish`** — EventBridge rule on the manager's approval event. Splits the rota per person, checks quiet hours, formats each person's own shifts from the voice template, attaches calendar invites, and ships via Slack incoming webhook (`ss/slack/webhook` in Secrets Manager) or SES `SendRawEmail`. On a quiet-hours defer, creates a one-off EventBridge Scheduler rule that re-invokes `publish` at the next reasonable minute. Writes rows to `ss-shifts` after a successful send. Memory: 256 MB. Timeout: 30 s.
- **`action-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on the request body. Triggered by Slack interactive clicks (Approve/Edit/Re-draft and the swap actions Cover/Drop/Time-off) and by email-link clicks. On approve, fires the publish event. On a swap, reuses the drafter's candidate logic to propose a replacement back to the manager, then on the manager's yes updates `ss-shifts` and `ss-hours`. Writes to `ss-audit` on every action. Memory: 256 MB. Timeout: 15 s.

- `digest` — EventBridge Scheduler target, weekly Sunday 6pm. Reads `ss-shifts` for the coming week; sends each person a short reminder of their upcoming shifts and the manager a heads-up on any still-open shifts. No Bedrock; the message is a plain summary table. Memory: 256 MB.
- `summary` — EventBridge Scheduler target, weekly Friday 5pm. Reads the week's `ss-hours` and `ss-audit`; calls Bedrock Haiku 4.5 to write a short fairness narrative (each person's hours against target, any drift); posts it to the manager's Slack. Memory: 512 MB.

Storage

- **DynamoDB** · `ss-shifts` — one row per published shift. PK `(week_id, shift_id)`; attributes: `day`, `start`, `end`, `role`, `assigned_to`, `status` (filled/open/held), `dispatched_via` (slack/email). On-demand. No TTL.
- **DynamoDB** · `ss-hours` — running placed-hours per person per week. PK `person_id`; sort key `week_id`; attributes: `hours_placed`, `hours_target`, `cap`. On-demand.
- **DynamoDB** · `ss-audit` — one row per write action of any kind. PK `(shift_id, ts)`; attributes: `action` (drafted/approved/swapped/dropped/timeoff), `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **DynamoDB** · `ss-shifts-archive` — archived weeks after they pass. Same shape as `ss-shifts`; PK `(week_id, shift_id)`. On-demand.
- **S3** · `ss-roster-source` — mirrored CSV from the Drive roster sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 3 years.

- **S3 · `ss-rules-source`** — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3 · `ss-raw-mime`** — raw inbound MIME from forwarded time-off notes. Lifecycle to Glacier at 30 days; expiry at 3 years.
- **S3 · `ss-published`** — a snapshot of each approved weekly rota as published, kept for reference and for re-sending a person their week on request.

Bedrock

- **Foundation model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global cross-Region inference profile `global.anthropic.claude-haiku-4-5-20251001-v1:0`. Two callsites: `intake-timeoff` for reading plain-English time-off notes, and `summary` for the weekly fairness narrative. The heavier `anthropic.claude-sonnet-4-6` profile is wired but unused by default — the tasks here are light enough for Haiku, and a model isn't on the hot path at all.
- **Embeddings.** Not used. The roster is structured rows; rule-based matching beats vector retrieval here. No Knowledge Base, no S3 Vectors.
- **Quotas.** Default account quotas are more than enough at small-team volume. The drafter itself doesn't call Bedrock; the time-off lane fires a few times a week at most.

EventBridge Scheduler config

- **`ss-weekly-draft`** — `cron(0 14 ? * 5 *)` (Thursday 2pm) in the team's timezone. Target: `drafter` Lambda.
- **`ss-drive-sync`** — `rate(15 minutes)`. Target: `drive-sync` Lambda.

- `ss-template-sync` — `cron(0 10 ? * 5 *)` (Thursday 10am, before the draft) in TZ. Target: `template-sync` Lambda.
- `ss-weekly-digest` — `cron(0 18 ? * SUN *)` in TZ. Target: `digest` Lambda.
- `ss-weekly-summary` — `cron(0 17 ? * 6 *)` (Friday 5pm) in TZ. Target: `summary` Lambda.
- **One-off rules** — created on the fly by `publish` when a quiet-hours defer is needed. Use `at(YYYY-MM-DDTHH:MM:SS)` expressions with `--action-after-completion DELETE` so the rule self-cleans.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `timeoff.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `ss-inbound-rules`: one rule with recipient `timeoff@your-company.com` → spam scan → S3 PUT to `s3://ss-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-timeoff`.
- SES outbound for the email-fallback schedules: verify a sender identity at `rota@your-company.com` with DKIM and SPF on the parent domain. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **drafter role:** `s3:GetObject` on the roster, rules, and voice keys; `dynamodb:Query` + `GetItem` + `PutItem` on `ss-hours`; `events:PutEvents`

on the default bus. *No* `bedrock:*`.

- **publish role:** `events:ListSchedules` + `CreateSchedule` for the deferred-publish one-offs; `secretsmanager:GetSecretValue` on the Slack webhook secret; `ses:SendRawEmail` from the verified sender identity; `dynamodb:PutItem` on `ss-shifts`; outbound network access to `hooks.slack.com`.
- **action-handler role:** `dynamodb:PutItem` on `ss-shifts`, `ss-hours`, and `ss-audit`; `secretsmanager:GetSecretValue` on the Sheets-API service-account secret; outbound network access to `sheets.googleapis.com`; `dynamodb:Query` for candidate lookup; `dynamodb:BatchWriteItem` for archiving a passed week to `ss-shifts-archive`.
- **intake-timeoff role:** `s3:GetObject` on `ss-raw-mime`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack webhook.
- **drive-sync and template-sync roles:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the roster and rules buckets; outbound network to `www.googleapis.com`.

Slack interactive flow

The Slack incoming webhook is the simplest delivery surface but doesn't support interactive button responses. So the manager-facing messages and the per-person schedules are posted via the `chat.postMessage` Web API instead, with Block Kit blocks containing the action buttons. Button clicks are sent by Slack to the configured Interactivity request URL, which is the `action-handler` Function URL. `action-handler` verifies the Slack signing secret on the inbound request, parses the action_id (`approve`, `edit`, `redraft`, `cover`, `drop`, `timeoff`),

opens a modal if needed (Edit and Cover open modals; Approve is one-tap), and processes the response when the modal is submitted.

The Slack app needs `chat:write`, `im:write`, and the Interactivity URL configured. The bot token lives in Secrets Manager under `ss/slack/bot-token`. The signing secret is `ss/slack/signing-secret`.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a CloudWatch metric for alerting.
- **Alarms:** drafter Lambda failures > 0 in a week (the weekly draft is the one piece that has to run); publish failure rate > 1% in 24h; action-handler signature-verification failures > 5/hour (might mean the Slack secret rotated).
- **X-Ray:** off by default. Not worth the cost at small-team volume.
- **AWS Budgets:** $15/month threshold, alarm at 80% and 100%, posts to SNS topic `ss-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for Drive, Sheets, and Calendar APIs all live in Secrets Manager under `ss/drive/sa` (one service account with scopes for all three APIs). Slack bot token, signing secret, and webhook URL all under `ss/slack/*`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, quiet-hours window, default max-hours cap, rest gap, and admin

fallback all live in Parameter Store under `/ss/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys) and AWS SAM. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for both `ss-roster-source` and `ss-rules-source` so a bad Drive edit can be rolled back in one click, and version the EventBridge Scheduler timezone setting so you don't accidentally start running the weekly draft in UTC after a CI rotation. SAM fits this surface well; CDK with a Python stack file also works. Total deployable surface: around eight Lambdas, four DDB tables, four S3 buckets, one EventBridge rule on the default bus (plus the Scheduler rules), one SES rule set, and one Budgets alarm.

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