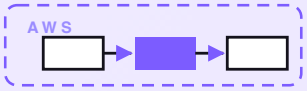
7-PART SERIES · FREE COMPANION



Menu sync

A serverless system that keeps a restaurant's menu and prices the same everywhere. The owner edits one master menu in a Drive sheet; every change — new item, price update, sold-out, seasonal special — flows out to the website, the online-order platforms, and the printable PDF; any place that rejects a change gets flagged. Big changes wait for a tap; routine price syncs can flow automatically within rules the owner sets. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Menu sync

PART 1 OF 7

JUNE 15, 2026 PART 1 OF 7 · MENU SYNC SERIES ~5 MIN READ

A menu sync on AWS for a few dollars a month

A restaurant's menu lives in more places than anyone keeps in their head. The website. The two online-order apps that take a cut of every order. The printable PDF guests download. The QR-code page on the table. When the kitchen runs out of the special, or a supplier raises the price of beef, or the owner adds a seasonal dish, every one of those places has to change — and in real life most of them don't. The website still lists a dish that sold out at lunch. One delivery app still charges last month's price. This post walks through the design of a small system where the owner edits one master menu and every change flows out to all the places the menu shows up — and any place that rejects a change gets flagged so somebody can fix it.

KEY TAKEAWAYS

- One master menu in a Drive sheet is the single source of truth; everything else is downstream.
- Every change ends in one of four moves on each run: in sync, push update, hold for approval, or flag a rejection.
- Routine price changes can flow automatically within rules the owner sets; big changes wait for a tap.
- Each place the menu shows up has its own adapter; a place that rejects an update is flagged, not forgotten.
- Designed on AWS for about $3/month at typical single-restaurant volume.

The whole system on one page

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

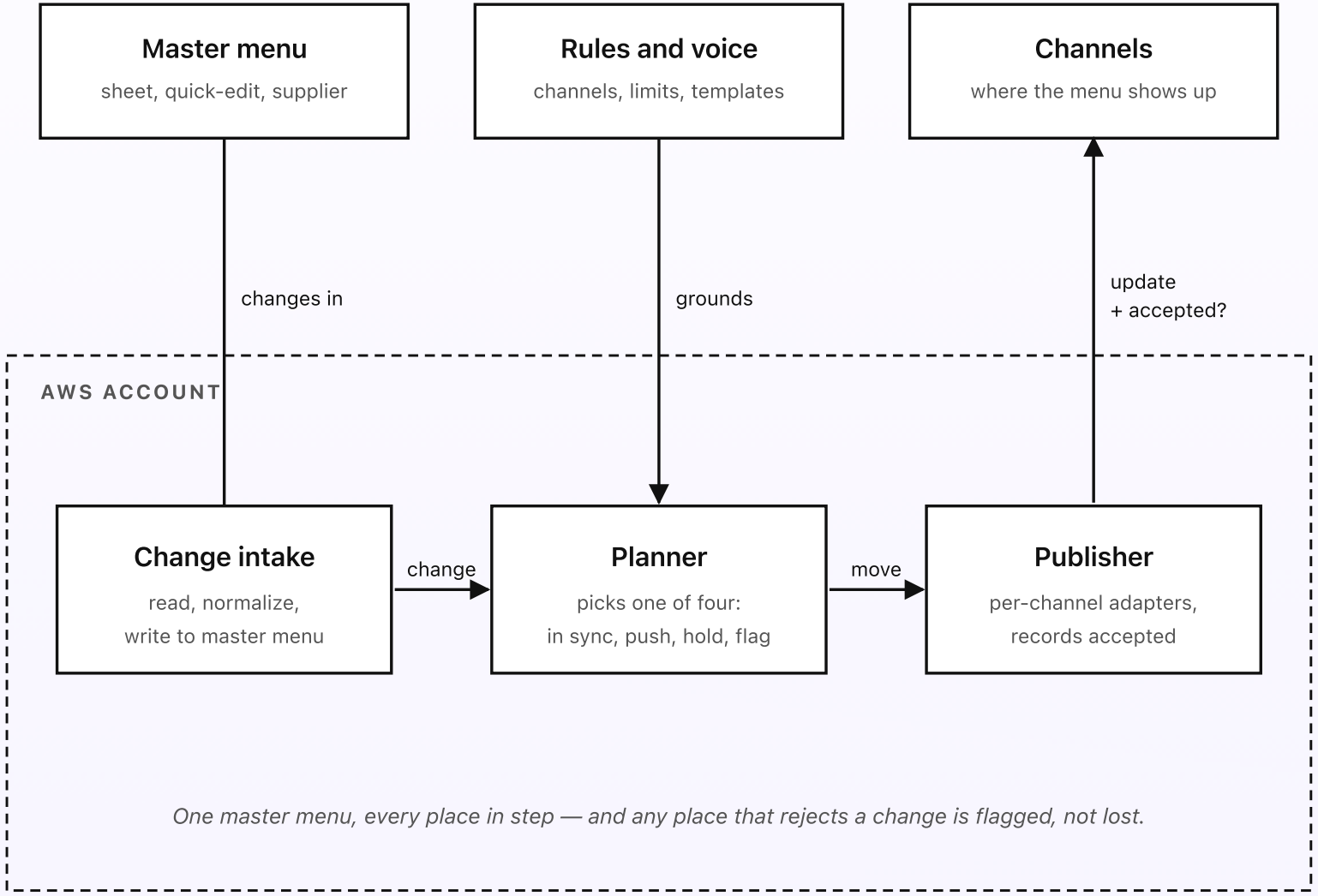

Master menu
sheet, quick-edit, supplier
Rules and voice
channels, limits, templates
Channels
where the menu shows up
changes in
grounds
update
+ accepted?
AWS ACCOUNT
Change intake
read, normalize,
write to master menu
change
Planner
picks one of four:
in sync, push, hold, flag
move
Publisher
per-channel adapters,
records accepted
One master menu, every place in step — and any place that rejects a change is flagged, not lost.

Fig 1. Three sources outside, three pieces inside AWS. Changes flow in from the master sheet, a quick-edit lane, and a supplier-price lane. The Planner runs on each change and picks one of four moves. The Publisher pushes the right update to each channel and records whether each one accepted it.

What you set up once (the outside)

- **Master menu.** A Google Sheet in a Drive folder, one row per item: name, description, category (starters, mains, drinks, specials), price, a sold-out flag, a seasonal flag, and which channels the item publishes to. You fill it in once and edit it whenever something changes. New changes can also enter via two other lanes covered in Part 2 — a quick-edit lane (mark a dish sold-out from your phone in two taps) and a supplier-price lane (forward a supplier price list and the system proposes the new costs for one-tap approval).
- **A rules folder.** Two short Google Docs in a Drive folder. The *rules* doc covers which channels each item category publishes to, the auto-sync limits for routine changes (for example: a price change under 10% can flow automatically; anything bigger waits for a tap), and the approval thresholds for risky moves like removing a whole category. The *voice* doc holds one formatting template per channel — how a dish name, description, and price should read on the website versus the printable PDF versus an order app that caps the description length.
- **Channels.** The places the menu shows up. Each channel has a small adapter that knows how to talk to it — the website's content store, each order platform's menu API, the PDF generator, the QR-code page. Each adapter reports back whether it accepted the update or rejected it (a price field it wouldn't take, an item name that's too long, a category the platform doesn't support).

What runs on every change (the inside)

- **The change intake.** Three sources feed the master menu. The Drive sheet itself is the canonical store. Changes can also be added via the quick-edit lane (tap a dish sold-out from a small phone page; the change lands in the sheet) and the supplier-price lane (forward a price list to `prices@your-restaurant.com`, the system uses Textract to read it and Bedrock Haiku 4.5 to match each line to a menu item and propose the new price, then drops a one-tap approval card in the owner's Slack before the price is written).
- **The planner.** Runs whenever the master menu changes. Reads the new menu. For each channel an item publishes to, compares what the master menu now says against what that channel currently shows. Picks one of four moves. *In sync*: the channel already matches — do nothing. *Push update*: the change is routine and within the auto-sync limits — send it straight to the channel. *Hold for approval*: the change is big (a price jump over the limit, a removed category) — hold it and ask the owner to approve before it goes out. *Flag a rejection*: a previous push was refused by the channel — surface it so somebody can fix the underlying problem. The planner itself doesn't call a model — the move logic is plain comparison.
- **The publisher.** Takes the chosen move and the channel's formatting template, formats the change for that channel, and sends it through the channel's adapter. Website and PDF changes go straight through; order-platform changes go through each platform's menu API. Every push writes a row in DynamoDB recording whether the channel accepted it, so the next run can tell what's in sync and what got rejected. A weekly digest summarizes what changed that week and which places are out of step. A monthly summary writes a short note:

how many changes flowed automatically, how many needed a tap, and which channel rejects most often.

In plain words

It's 11am and the kitchen tells you the slow-braised short rib is 86'd for the day — sold out. You open the quick-edit page on your phone and tap the dish sold-out. Within a minute the website grays it out, both order apps mark it unavailable so no guest can order something the kitchen can't make, and the QR-code page updates. The printable PDF regenerates overnight. Later that week your beef supplier emails a new price list; you forward it to the prices address. The system reads it, matches the short rib's cost to the line on the sheet, and proposes a $2 price bump. Because $2 is under your 10% auto-sync limit, you tap approve and it flows everywhere. The one order app that rejected the change — its API wanted the price in cents, not dollars — shows up flagged in your Slack so it gets fixed once instead of silently charging the old price for a month.

The cost of running this is about $3 a month at single-restaurant volume. The cost of *not* running it is the guest who orders a sold-out dish and waits twenty minutes for an apology, or the delivery app quietly selling your food below cost because nobody updated the price.

DESIGN RULES THAT SHAPED EVERY DECISION

- One master menu is the only source of truth. Every channel is downstream; none of them is edited directly.
- Four moves, always. In sync, push update, hold for approval, flag a rejection. There is no fifth.
- Routine changes flow automatically within the owner's rules; risky ones wait for a tap. Nothing big goes out unseen.
- A rejected update is flagged, never swallowed. A place that's out of step is visible, not silent.
- The menu lives in Drive. Changing a price, marking sold-out, or adding a dish doesn't need a deploy.
- Every push is logged. Ask "why does this app still show the old price?" next month and you can see exactly what happened.

Why this shape

Most restaurants keep their menu in one of three ways: a single place they treat as real (usually the website) and a mess of copies they update by hand, a stack of separate logins they remember to touch only when a guest complains, or somebody's memory of which app has which price. The hand-updated copies drift the moment a busy night arrives. The separate logins are exactly the chore that gets skipped first. And memory fails the day the person who held it is off and the kitchen runs out of the special.

The setup above moves the source of truth into a sheet the owner already edits, but adds a small system that *watches* that sheet and pushes each change out to every place the menu lives. Routine changes — sold-out, a small price tweak — flow on their own within rules the owner set. Big changes wait for a tap so nothing surprising ships. And when a channel refuses a change, the system says so instead of pretending it worked. The sync is invisible on the days nothing changes; it only shows up when something does — and on the days a channel pushes back.

The next four posts walk through each piece in turn: how a menu change gets made, how a change turns into per-channel updates, how an update reaches each place, and how a rejected update gets fixed. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 15, 2026 PART 2 OF 7 · MENU SYNC SERIES ~4 MIN READ

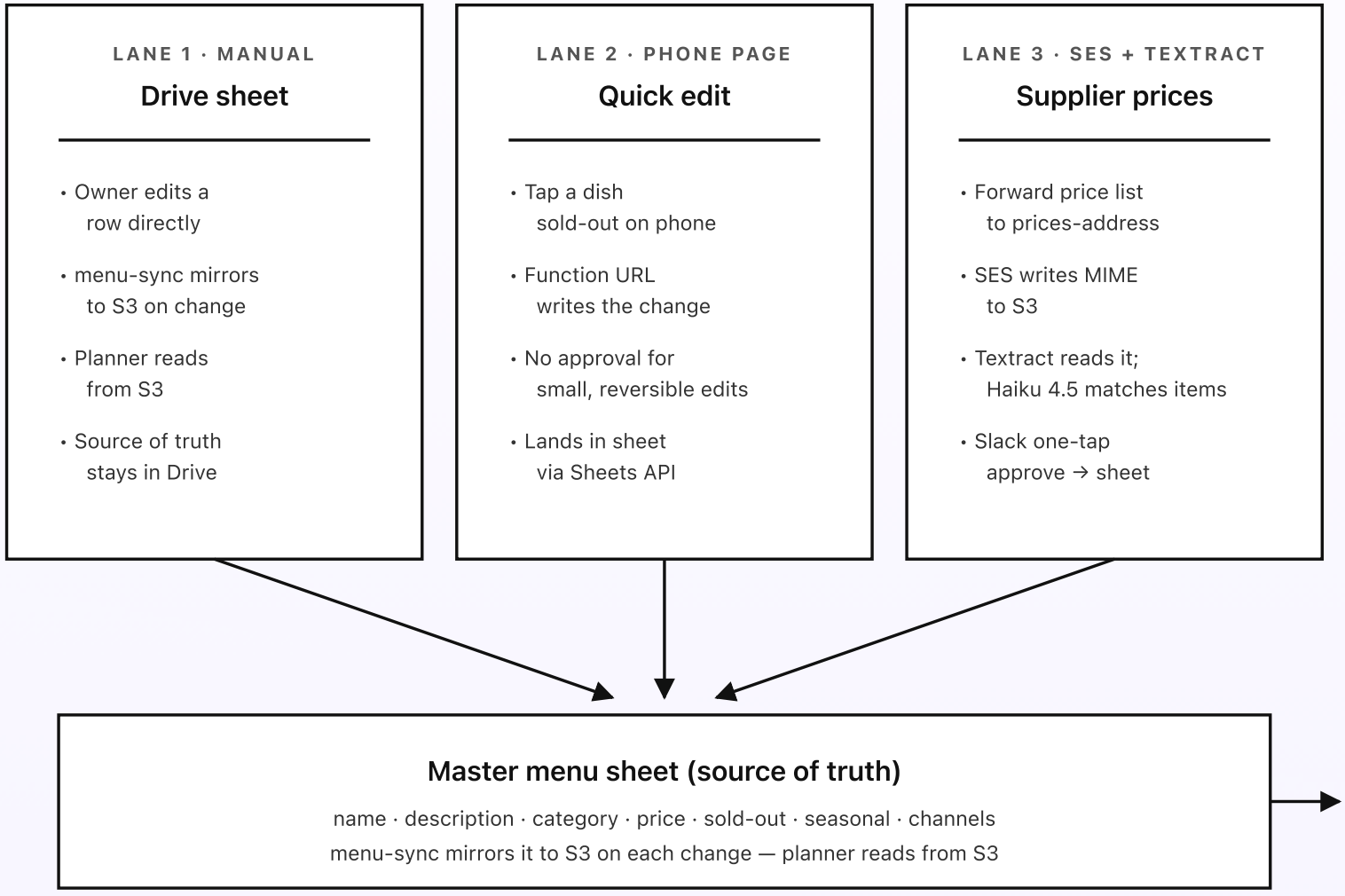
How a menu change gets made

The sync only pushes what’s in the master menu. So the first job is making it easy to change the master menu, wherever the owner happens to be. There are three ways a change gets in: somebody edits the Drive sheet, somebody taps a dish sold-out on a phone page, or somebody forwards a supplier price list to a dedicated address. The first one is obvious. The other two exist because in real life nobody opens a spreadsheet on a busy lunch service to mark the special 86’d.

KEY TAKEAWAYS

- Three intake lanes feed one master menu: the Drive sheet, a phone quick-edit lane, and a supplier-price lane.
- Forwarded price lists are read by Textract; Bedrock Haiku 4.5 matches each line to an item and proposes a price.
- Every proposed price change goes to the owner's Slack for one-tap approval before it lands in the menu.
- The quick-edit lane is a tiny phone page for the two changes that happen mid-service: sold-out and small tweaks.
- The Drive sheet stays the canonical store. The other lanes are conveniences that write into it.

Three lanes into one master menu



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

Fig 2. Three lanes converge on one master menu sheet. The sheet is the source of truth; the phone lane and the supplier lane are conveniences that write into it. The menu-sync Lambda mirrors the sheet to S3 so the planner can read it without hitting Drive on every change.

Lane 1: the Drive sheet itself

The simplest lane. Open the master menu sheet in Drive, edit a row, save. The columns are short: name, description, category, price, a sold-out flag, a seasonal flag, and a small list of which channels the item publishes to. A change-triggered Lambda — `menu-sync` — exports the sheet as plain JSON via the Drive API and writes it to `s3://ms-menu-source/menu.json` whenever the sheet changes. The planner 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 you're sitting down with the menu in front of you — adding a new seasonal dish, rewriting a description, reorganizing categories. Most planned menu work goes in this way.

Lane 2: the quick-edit page (the lane staff actually use mid-service)

Two changes happen in the middle of a busy service: a dish sells out, or a price needs a quick nudge. Nobody is opening a spreadsheet for those. So the quick-edit lane is a tiny phone web page — a list of every item with a sold-out toggle and a price field. Tap the short rib sold-out and the page posts to a small `quick-edit` Function URL Lambda, which writes the change straight to the Drive sheet via the Sheets API and triggers the planner.

These changes don't need approval. Marking a dish sold-out is reversible (toggle it back when the kitchen restocks), and a small price nudge inside the auto-sync limit is exactly the kind of routine change the rules already allow to flow. The page is locked behind a simple staff PIN stored in Parameter Store, so only the floor team can reach it. The point of this lane is speed: the change a server makes in two taps should be live everywhere before the next table orders.

Lane 3: supplier prices

Costs change when suppliers change them, and they almost always arrive as a PDF or spreadsheet attached to an email. Forcing the owner to retype a price list into the menu sheet is a chore that gets skipped — and skipping it means selling food at last quarter's cost.

Lane 3 picks up forwarded price lists. Set up a dedicated address — something like `prices@your-restaurant.com` — via Amazon SES. Forward a supplier list to it and the system takes over. SES writes the raw MIME to `s3://ms-raw-mime/`. The S3 PUT triggers a parser Lambda that runs Amazon Textract on the attachment to read the lines, then calls Bedrock Haiku 4.5 to match each supplier line to a menu item and propose an updated cost. The prompt is short: "Match each line to a menu item by name. Return JSON only. Don't invent a match you're unsure of — mark it for review instead." The proposals go to the owner's Slack with *approve*, *edit*, and *discard* buttons. On approve, the new prices are written to the sheet.

Every proposed price goes to a human first for a simple reason: a price the model misread is worse than a price that never updated at all. The misread one quietly charges the wrong amount everywhere until somebody notices on the month-end numbers.

Why the menu stays the source of truth

Three lanes in, but only one place the planner actually reads. That's a deliberate constraint. If two lanes both wrote directly to the channels, every "why does the website say this?" question would mean checking three places. Funneling everything through the Drive sheet means there is exactly one row per item, and any staff member can read or edit it without learning a new tool. The convenience lanes are first-class for getting changes in, but they always pass through the sheet on the way.

Next post: how the planner reads the master menu, compares it against what each channel currently shows, and picks one of four moves per channel.

PART 3 OF 7

JUNE 15, 2026 PART 3 OF 7 · MENU SYNC SERIES ~5 MIN READ

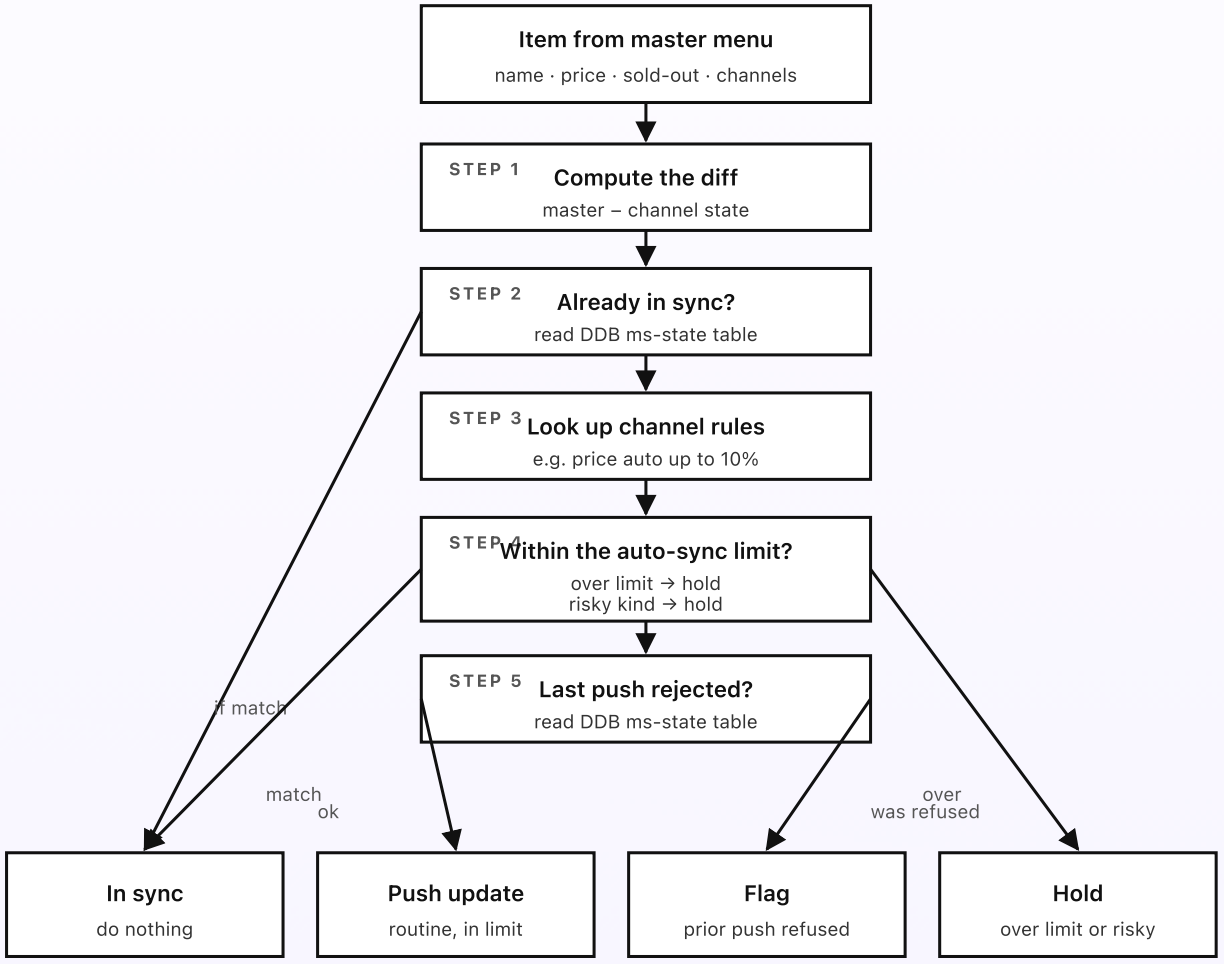
How a change turns into updates

A change lands in the master menu — a price moved, a dish went sold-out, a new special appeared. The planner Lambda wakes up, reads the menu, and looks at one item-and-channel pair at a time. For each one it asks: does this channel already match? If not, is the change small enough to push on its own, or big enough to need a tap? The whole decision is plain comparison. No model. Every limit lives in the rules doc, where the owner can edit it without a deploy.

KEY TAKEAWAYS

- The planner runs on each change, triggered by the menu mirror landing in S3.
- Auto-sync limits live in the rules doc — for example a price change under 10% flows on its own; bigger waits for a tap.
- Four moves per item-and-channel: in sync, push update, hold for approval, flag a rejection.
- DynamoDB tracks what each channel currently shows so the planner only acts on real differences.
- The planner itself never calls a model. The decision is entirely a comparison against the rules.

The decision flow, per item and channel



The rules doc holds every limit — change one and the next change uses the new value.

Fig 3. The planner's decision tree, per item, per channel, on each change. Five steps decide which of four moves applies. The rules doc holds every limit; the planner only enforces them.

Auto-sync limits: 10% isn't magic, it's in the doc

The rules doc has one short section per kind of change. Each section names the limit in plain prose: "Price changes: flow automatically up to 10%; bigger waits for approval. Sold-out and back-in-stock: always automatic. New dishes: always hold for approval. Removing a whole category: always hold." The numbers are the line between "push it" and "ask first." Routine changes that the owner has decided are safe flow on their own; anything that could surprise a guest or hurt the numbers waits for a tap.

The limits exist for a reason. A small price tweak is the kind of thing that happens weekly and shouldn't need the owner's attention each time. A 30% jump almost always means a typo or a real decision that deserves a second look before it's charged everywhere. Marking a dish sold-out is reversible and urgent, so it's always automatic. Removing a category is rarely reversible in a hurry, so it always waits. Different changes carry different risk; the limits reflect that.

Per-channel overrides exist too. The rules doc can set a tighter limit for one channel — maybe the delivery app that charges guests a service fee should never auto-accept any price change, so every price move there is held no matter how small. That's the right escape hatch for the one place where mistakes cost the most.

Four moves, always

Every item, every channel, every change lands in exactly one of four buckets. The names are simple on purpose.

- **In sync.** The channel already shows what the master menu says. Do nothing. Most items, on most changes, are already in sync on most channels.
- **Push update.** The change is routine and inside the auto-sync limit. Send it straight to the channel through its adapter. Write a row to the `ms-state` DynamoDB table marking that this push went out.
- **Hold for approval.** The change is over the limit or a risky kind — a big price jump, a new dish, a removed category. Hold it and send the owner a one-tap approval card. Nothing ships to the channel until they approve.
- **Flag a rejection.** A previous push to this channel was refused — a price field it wouldn't take, a name too long, a category it doesn't support. Surface it so somebody can fix the underlying problem. Mark the item-and-channel as flagged in DynamoDB; it stays flagged until the next successful push or a manual clear. Part 5 covers what fixing a rejection actually looks like.

State that makes the decision deterministic

The planner reads one DynamoDB table on each change. `ms-state` records the last accepted value per item per channel: `(item_id, channel, last_value, last_push_status, last_push_at)`. With that one table, the move decision is a few dozen lines of comparison and zero magic. A given item with a given master value and a given channel state always produces the same move. Re-running the

planner produces no extra pushes — the state shows what each channel already has, so an item already in sync stays in sync.

When a push succeeds, the publisher updates `ms-state` with the new accepted value. When a push is rejected, it records the rejection and the reason. The planner reads both on the next change. That's the whole memory the system needs to tell "in sync" from "needs a push" from "something's broken."

Why the planner uses no model

The planner could call a model to decide whether a change is "big" or to rewrite a description for a channel. It doesn't. Two reasons. First, the planner should be the one part of the system that is utterly predictable — if the rules say a price under 10% flows and a bigger one holds, that's exactly what happens. A model in that loop introduces variance the owner can't reason about, on the part of the system that touches what guests are charged. Second, model calls cost money, and the planner runs on every change, so the call would mostly be wasted on items that are already in sync.

Bedrock fires elsewhere — on the supplier-price lane in Part 2, and on the monthly summary mentioned in Part 6. Not in the planner. The planner itself is plain comparison that reads a menu and writes events.

Next post: how an update reaches each place — the per-channel adapters, the formatting templates, and the guardrails between the planner's chosen move and the change actually landing.

PART 4 OF 7

JUNE 15, 2026 PART 4 OF 7 · MENU SYNC SERIES ~5 MIN READ

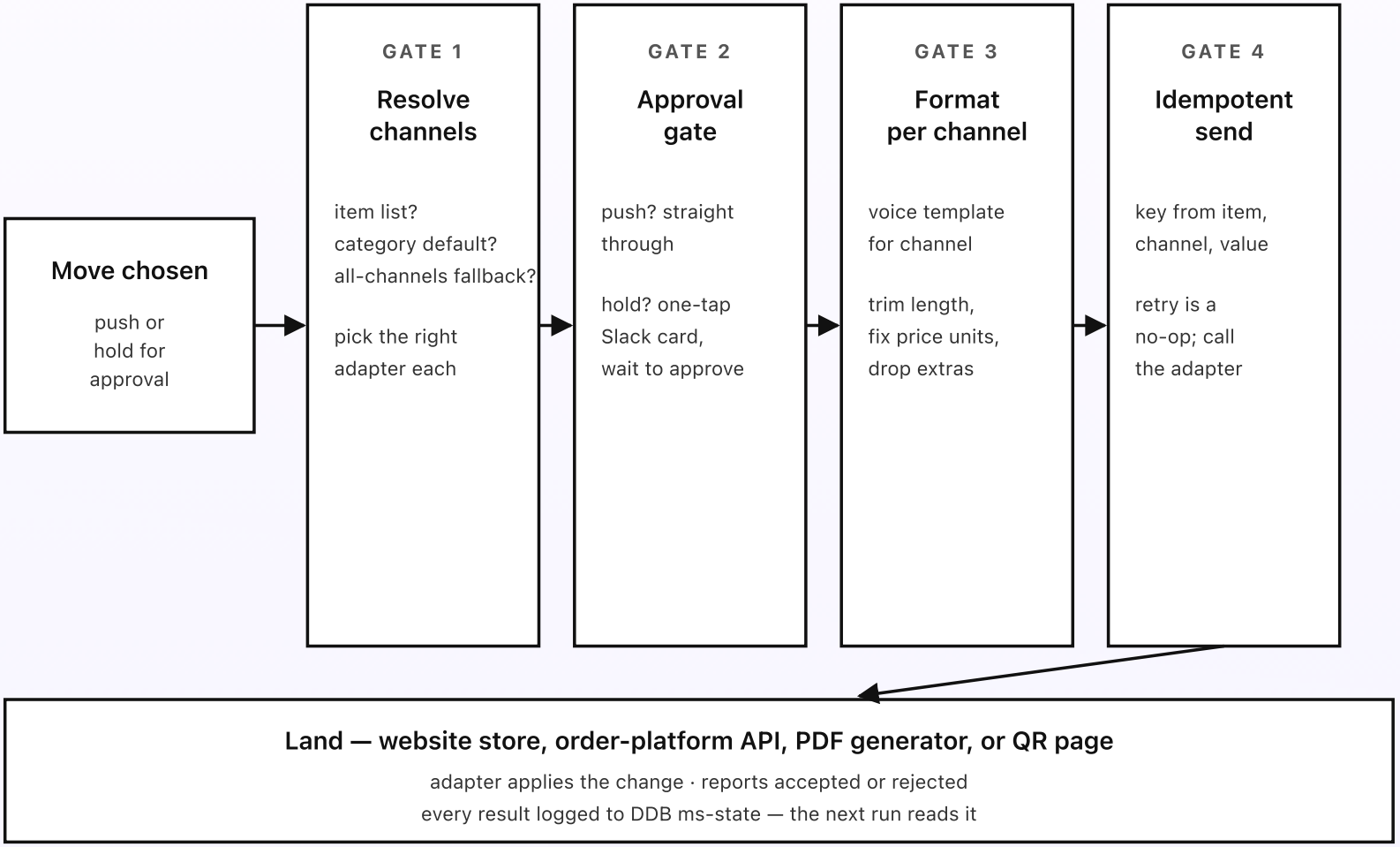
How an update reaches each place

The planner picked a move — push or hold. Now the publisher Lambda has to turn that into a real change on a real channel: the right format for that place, an owner's tap when one is needed, the right call to the channel's adapter, and a record of whether it worked. Get any of those wrong and the update is worse than no update: a price in dollars where the app wanted cents, a description too long for the website, the same change pushed twice. Four small guardrails sit between the move and the change landing.

KEY TAKEAWAYS

- Channel resolution: a per-item channel list beats the per-category default beats the all-channels fallback.
- Routine pushes flow on their own; held changes wait for the owner's one-tap approval first.
- Each channel has an adapter that formats the change and speaks that channel's own language.
- Every push carries an idempotency key, so a retry never applies the same change twice.
- The result — accepted or rejected, with the reason — is written to DynamoDB for the next run to read.

Four guardrails on every push



Every gate is a deterministic check — no model calls, and a retry never charges a guest twice.

Fig 4. Four guardrails between the move and the landed update. Resolve the channels. Wait for approval on held changes. Format for the channel. Send idempotently. Then apply via the adapter and log the result so the next run knows the state.

Gate 1: resolve the channels

Three places the publisher looks to decide which channels an item belongs on, in order. First, the item's own `channels` column in the master menu — if a dish is set to publish only to the website and the PDF (a dine-in-only special that shouldn't appear on delivery apps), that list wins. Second, the per-category default in the rules doc ("all drinks publish everywhere except the delivery apps"). Third, the all-channels fallback — if nothing is set, the item goes everywhere. The fallback is the safe default for a normal dish.

For each channel in the resolved list, the publisher picks the right adapter. An adapter is a small piece of code that knows how to talk to one channel: the website's content store, a specific order platform's menu API, the PDF generator, the QR-code page. Adding a new channel later is writing one new adapter, not changing the rest of the system.

Gate 2: the approval gate

If the planner's move was *push*, this gate is a pass-through — the change was already judged routine and inside the limits, so it flows straight on. If the move was *hold*, the publisher sends the owner a one-tap approval card in Slack: here's the item, here's the old value and the new value, here's which channels it'll touch, with Approve and Reject buttons.

Nothing ships to any channel until the owner approves. If they approve, all four gates run as if it were a push from the start. If they reject, the path ends cleanly — the master menu keeps the change (it's still the source of truth), but the channels are left as they were, and the held item shows up in the weekly digest so it isn't forgotten. This is the human-in-the-loop guardrail: no surprising change to what a guest is charged ever ships without a person seeing it first.

Gate 3: format for the channel

Channels disagree about everything. The website can show a long, lyrical description; an order app caps it at 120 characters. The website wants a price like "$18.50"; one platform's API wants the integer `1850` in cents. The PDF wants a category header; the QR page wants a flat list. Gate 3 runs the change through that channel's formatting template from the voice doc: trim the description to the cap, convert the price to the channel's units, drop any field the channel doesn't support, and map the category name to the one that channel expects.

The templates are plain text with placeholders, edited in the voice doc without a deploy. They're the one place that encodes "this is how the short rib reads on the delivery app versus the printed menu," so the rest of the system can stay channel-agnostic.

Gate 4: send it once, even on a retry

The last gate attaches an idempotency key — a short fingerprint built from the item, the channel, and the exact value being sent — before calling the adapter. (Idempotent just means "safe to do twice": if the same change is sent again,

nothing extra happens.) If a network blip makes the publisher retry, the adapter sees the same key it already processed and treats the second call as a no-op instead of applying the change a second time.

This matters most for price changes. Without the key, a retry during a flaky moment could bump a price twice, or flip a sold-out flag back and forth. With it, the worst a retry can do is confirm what already happened. The adapter calls the channel — the website content store, the platform's menu API, the PDF regenerate job, the QR page — and reports back accepted or rejected, with the reason on a rejection. The publisher writes that result to `ms-state` in DynamoDB. The next run reads it and knows the true state of every channel.

Why the guardrails exist

None of these gates are exotic. They're the kind of small care a careful manager would take if they were updating each app by hand — check which places this dish even belongs on, get a second pair of eyes on the big changes, format it the way each app expects, and don't fat-finger the same price twice. Putting them in code as four small sequential gates makes them part of the design, not a habit you're trusting a busy person to keep on a busy night.

Next post: what happens when a channel says no — how a rejected update gets flagged, surfaced, and fixed, and how the system keeps the menu honest about which places are out of step.

PART 5 OF 7

JUNE 15, 2026 PART 5 OF 7 · MENU SYNC SERIES ~5 MIN READ

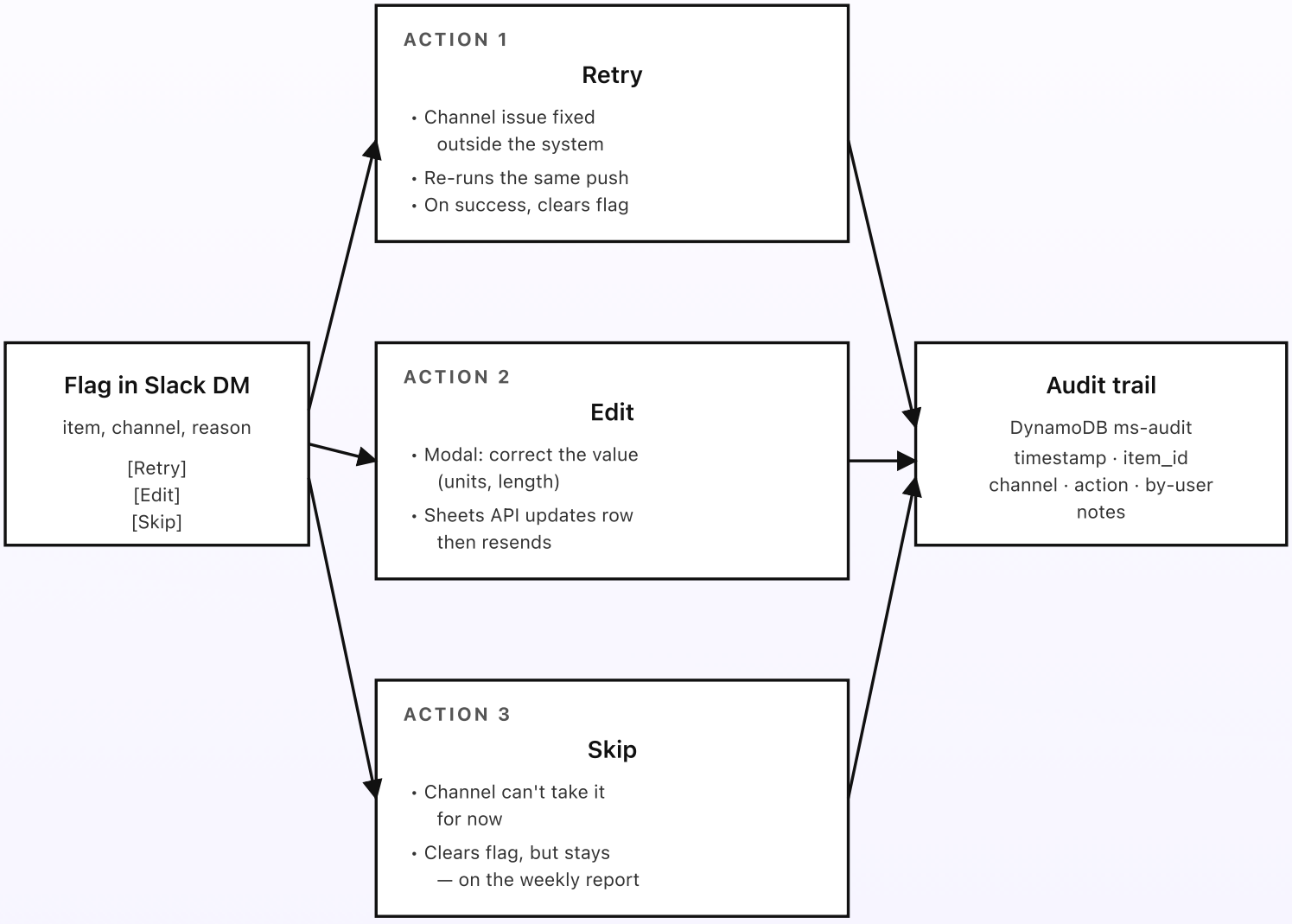
How a rejected update gets fixed

A flag lands in the owner's Slack at 11:04am. The delivery app rejected the short rib's new price. There's a Fix button. What happens when they tap it? The honest answer is "it depends on why the channel said no." This post walks through the three things the system can do on a flagged rejection — retry, edit, or skip — and how the master menu, the channel state, and the audit trail all stay in sync.

KEY TAKEAWAYS

- Three actions per flag: *retry* (resend after a fix), *edit* (correct the value and resend), *skip* (accept the channel can't take it).
- Each action updates the channel state via the adapter and writes an audit row.
- A successful retry clears the flag; the channel is back in sync with the master menu.
- Skip is bounded — a long-skipped item keeps showing on the weekly out-of-step report so it isn't forgotten.
- The Fix button is a Slack interactive message backed by a Function URL.

Three actions on a flag



Skip doesn't pretend the channel is in sync — it stops the noise. The weekly report still lists it.

Fig 5. Three actions per flag, three different effects. Retry resends after a fix. Edit corrects the value and resends. Skip accepts the channel can't take it for now. Every action writes to the audit trail.

Action 1: retry (the "it's fixed now")

Most rejections aren't about the value at all — they're about the channel. A delivery platform's access token expired, so every push bounced. The platform had a short outage during lunch. A network hiccup dropped the call. In all of these, the value the system tried to send was perfectly correct; the channel just wasn't ready to take it.

When the owner has fixed the underlying thing — re-authorized the token, waited out the outage — they tap *Retry*. The Fix button submits to a Function URL Lambda, which re-runs the exact same push through the adapter, carrying the same idempotency key from Part 4. Three things happen, in order. First, the adapter applies the change and reports success. Second, the `ms-state` row for that item-and-channel is updated to the new accepted value and the flag is cleared. Third, an `action: retried` row is written to `ms-audit` with the user, the timestamp, and the result. The channel is now back in sync with the master menu, and the next run sees it as in sync.

Action 2: edit (the value was wrong)

Sometimes the channel was right to say no. The price went out in dollars where that platform's API wanted cents. A new dish name ran past the channel's length

cap. A category name didn't match anything the platform supports. These are real problems with the value, and a plain retry would just bounce again.

Edit opens a small Slack modal pre-filled with the rejected value and the channel's reason, so the owner can see exactly what to fix. They correct it — usually a one-character change — and hit Save. On save, the Function URL Lambda writes the corrected value to the master menu via the Sheets API (so the fix lives in the source of truth, not just on one channel), then resends through the adapter with a fresh idempotency key. If the correction belongs only on that one channel — a shorter name just for the app that caps length — the edit updates that channel's formatting template in the voice doc instead, so future changes are formatted right automatically. Either way, the flag clears on the successful resend and an `action: edited` row lands in the audit trail.

Action 3: skip (the channel just can't take it)

Sometimes a channel genuinely can't hold an item and never will. The delivery platform doesn't support a "chef's table" category. A dish is only sold dine-in and shouldn't be on the app at all. A seasonal special is ending tomorrow and isn't worth chasing onto every channel. The owner doesn't want to keep being nagged about a rejection they can't resolve.

Skip writes a row to `ms-state` marking the item-and-channel as `skipped` with the date and the owner who skipped it. The flag clears so it stops pinging. But — and this is the important part — the system doesn't pretend the channel is in sync. The skipped item still shows on the weekly out-of-step report, with a small note ("skipped by owner on 2026-06-17"), so a genuine drift never silently disappears.

If the item is later set to stop publishing to that channel in the master menu, the skip resolves naturally and the report drops it. Skip stops the noise without losing the truth.

Every action is logged, every action is reversible

The `ms-audit` table records every retry, edit, and skip with the user who took the action, the timestamp, the channel, and a snapshot of the value before and after. If a wrong correction gets entered — a price fixed in the wrong direction, a name shortened too far — the owner can run an “undo last action” through a small admin command that reads the previous-state snapshot and restores both the master menu row and the channel. The undo is itself an audit row, so the trail of edits stays clean.

This kind of reversibility matters because the system touches what guests are charged across several platforms at once. The weekly out-of-step report is the safety net: at any moment, the owner can see exactly which places match the master menu and which don’t — and for the ones that don’t, whether it’s a bug to fix or a skip they chose.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at single-restaurant volume; Part 6 explains exactly where the dollars go and why most of the bill is the work of pushing changes, not thinking about them.

PART 6 OF 7

JUNE 15, 2026 PART 6 OF 7 · MENU SYNC SERIES ~3 MIN READ

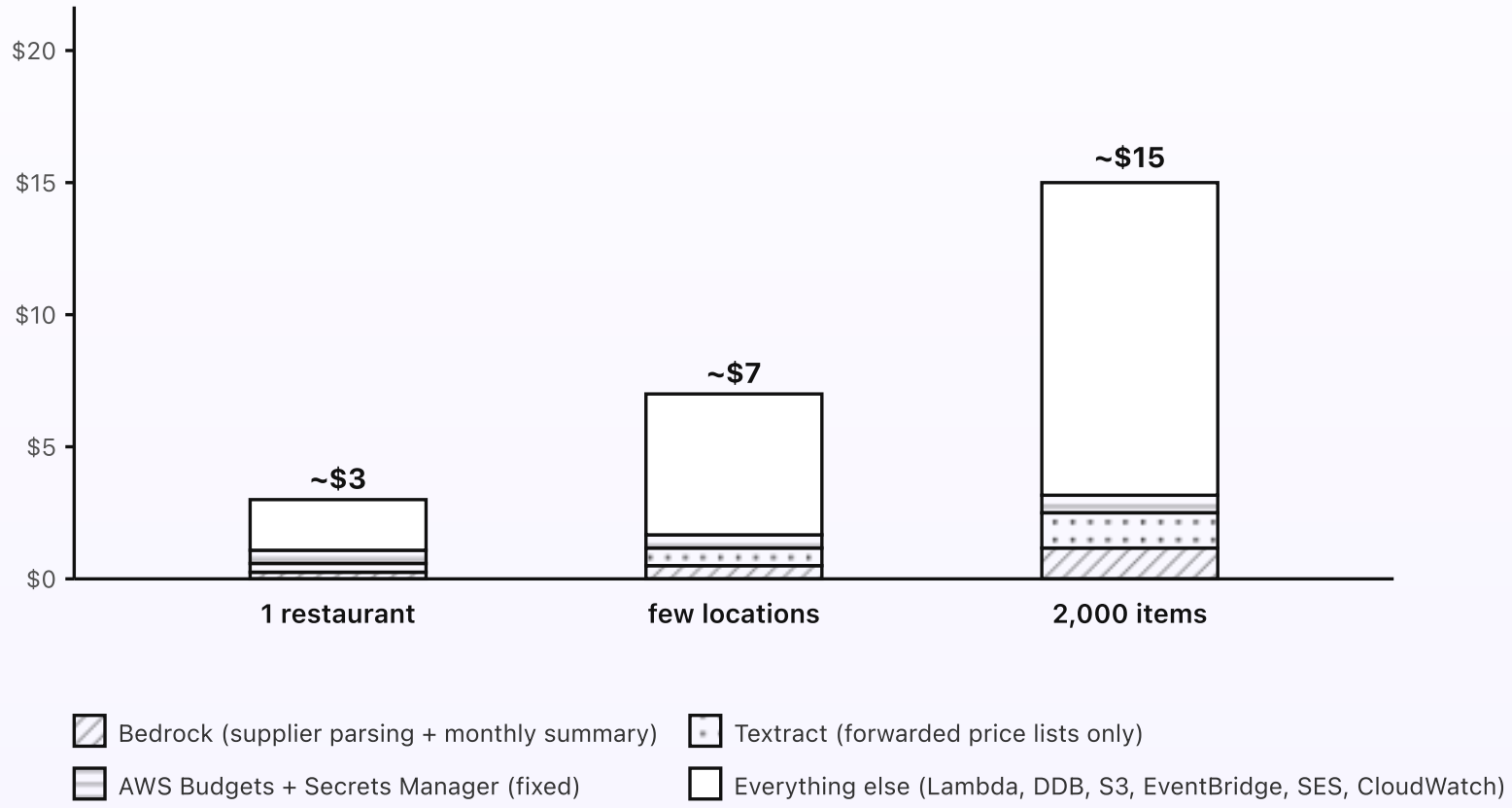
What the menu sync costs

The sync is one of the cheaper systems in this whole series. It only does work when the menu changes — it reads the new menu from S3, compares it against the channel state, and makes a handful of calls to push the differences. It calls no models in the planner. Bedrock fires only when somebody forwards a supplier price list and once a month for the owner's summary. At single-restaurant volume, the bill is a few dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $3/month at single-restaurant volume (around 120 menu items across the usual channels).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The planner costs pennies — no model calls, and it only runs on a change.
- Bedrock fires only on supplier-price parsing (a few times a month) and the monthly summary.
- At 500 items across a few locations the bill is around $7. At 2,000 items it's around $15.

Cost at three volumes



The push work is the dominant cost — and even that is fractions of a cent per channel per change.

Fig 6. Monthly cost at three menu volumes. Bedrock and Textract are small slivers because they only fire on the supplier-price lane and the monthly summary. The dominant cost is the everything-else bucket: the work of pushing each change to each channel.

Where the dollars actually go

Lambda runtime (the bulk). The planner runs only when the menu changes — a sold-out toggle, a price tweak, a new dish. Each run reads the menu JSON from S3, compares it against the channel state, and emits an event per channel that needs a push. That's milliseconds. The publisher then fires once per channel per change to apply it. A normal restaurant makes a few dozen changes a day at most across all channels; the Lambda total still lands well under a couple of dollars at single-restaurant volume. Add the quick-edit Function URL, the menu-sync mirror, and the supplier parser, and Lambda is still the cheap part.

DynamoDB on-demand. Two small tables: `ms-state` (the last accepted value per item per channel) and `ms-audit` (every action). Reads happen on each planner run; writes are one per push and one per fix. Pennies a month at any of these volumes.

S3 + Storage. The mirrored menu JSON, the generated PDFs, and the archived MIME from any forwarded price lists. A few MB total at SMB volume. Effectively free.

EventBridge. The change events from the planner to the publisher, plus the scheduled digest and summary rules. A few hundred events a day. Pennies.

SES. Inbound for the supplier-price lane: $0.10 per thousand received messages (so a couple of cents a year for one restaurant). Outbound for the weekly digest and monthly summary emails: $0.10 per thousand sent. Both are negligible at this scale.

Bedrock (only when something fires it). The planner uses no Bedrock. The supplier-price lane fires Haiku 4.5 once per forwarded list: a few thousand input tokens (the Textract output) and a few hundred output tokens (the matched-price JSON), so a fraction of a cent per parse. At a few price lists a month, Bedrock costs cents. The monthly summary is one larger call: write a paragraph that summarizes the month's changes, auto-syncs, and rejections; a couple of cents.

Textract (only on forwarded price lists). Per-page pricing; a typical supplier list is one to five pages. A few cents per parse. At a few lists a month, Textract is a few cents to a dollar. At chain volume with twenty lists forwarded per month, it lands around a dollar or two.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the quick-edit page and the Slack fix buttons.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The system does nothing between changes.
- **A Knowledge Base.** The menu is structured rows, not free text — deterministic comparison beats vector search here. No embeddings, no Knowledge Base, no S3 Vectors needed.
- **Models in the planner.** The decision is plain comparison. Bedrock fires only on the supplier-price lane and the monthly summary.

How the cost scales

Lambda runtime grows with the number of pushes, not the number of items sitting still — an item nobody changes costs nothing to keep in sync. So the bill tracks how often the menu changes and how many channels each change touches. DynamoDB grows the same way. Bedrock and Textract are uncorrelated with item count — they only fire when somebody forwards a price list or it's the first of the month. So a busy chain with 2,000 items and many channels lands around $15; a quiet single restaurant lands around $3. Past chain volume you'd batch pushes per channel to cut the per-call overhead, but that's an optimization for very large groups — not a redesign.

Set an AWS Budgets alarm at $25/month so anything unusual pages you before the bill matters. The sync's normal-volume 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 the channel-adapter contract.

PART 7 OF 7

JUNE 15, 2026 PART 7 OF 7 · MENU SYNC SERIES ~8 MIN READ

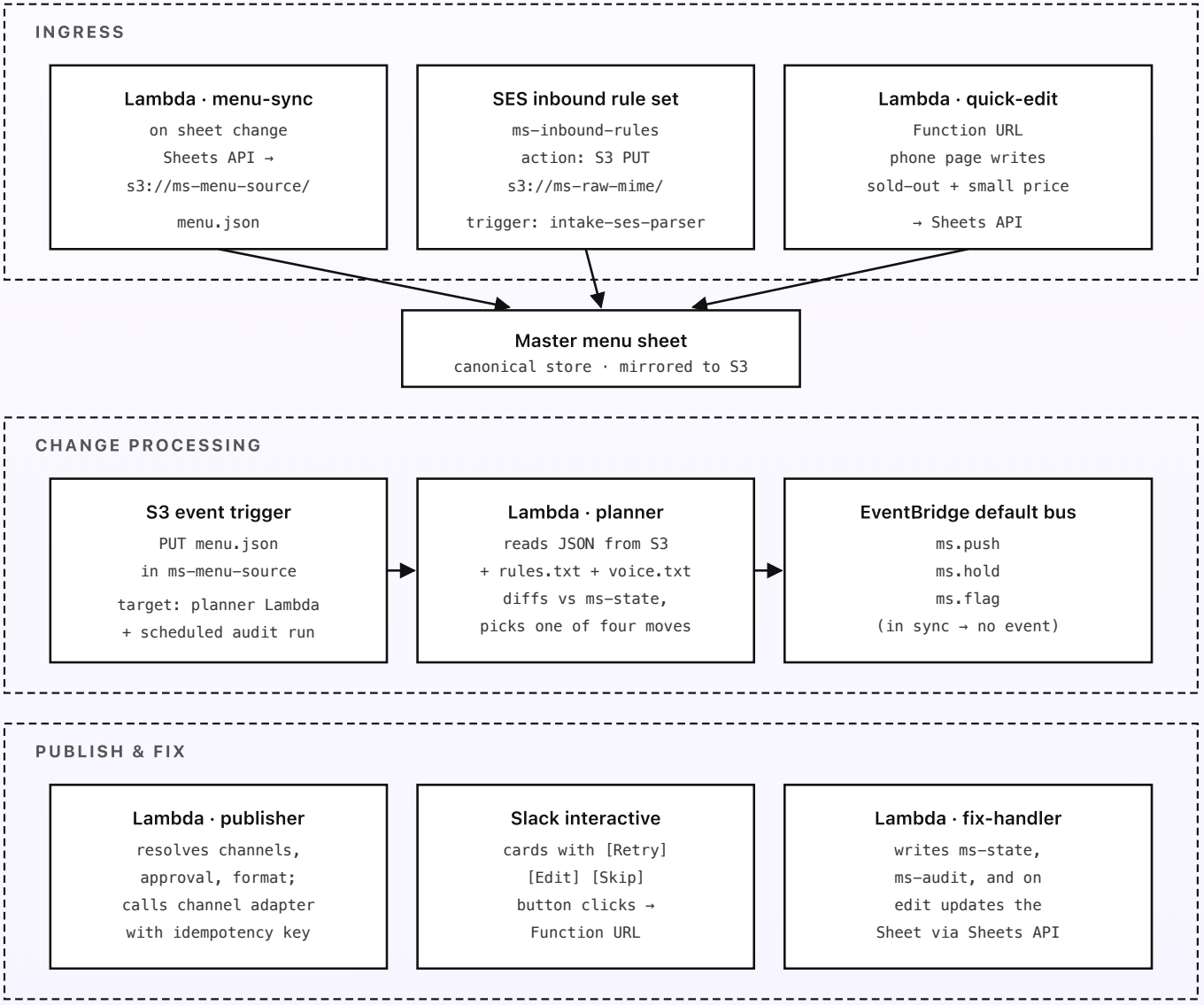
Engineering reference: the menu sync architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES inbound rule set, EventBridge config, the DynamoDB schemas, the channel-adapter contract, 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 are all in good shape there. A second region for multi-region resilience isn't worth the extra setup work at SMB volume — the failure mode for a restaurant is one channel showing a stale price for an hour, not a regional outage. One AWS account dedicated to the sync (separate from your other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



One master menu, every channel in step — and every interaction is logged to ms-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the menu), change processing (the planner diffing and emitting events), publish and fix (the update lands and a rejection's fix is recorded). Every Lambda is event- or trigger-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.

- **`menu-sync`** — triggered on sheet change (Drive push notification to a Function URL, with a fallback EventBridge Scheduler poll every 5 minutes). Uses the Google Drive API + Sheets API (service-account credentials in Secrets Manager under `ms/drive/sa`) to export the master menu sheet as JSON and write to `s3://ms-menu-source/menu.json` only if the sheet has changed since the last sync. The same pattern mirrors the rules and voice docs to `s3://ms-rules-source/`. Memory: 256 MB. Timeout: 30 s.
- **`quick-edit`** — Lambda Function URL serving a small static phone page plus its write endpoint. Gated behind a staff PIN from Parameter Store. Writes sold-out toggles and in-limit price changes straight to the Drive sheet via the Sheets API, then lets the normal `menu-sync` mirror + planner flow run. Memory: 256 MB. Timeout: 15 s.
- **`intake-ses-parser`** — S3 PUT trigger on `s3://ms-raw-mime/`. Parses MIME, extracts the price-list attachment, runs Textract via

`StartDocumentTextDetection` + `StartDocumentAnalysis` (asynchronously, to handle multi-page lists and tables). On Textract completion (via SNS notification), reads the structured 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 match each line to a menu item by name and propose updated prices. Posts the proposal to Slack via the incoming webhook with Approve/Edit/Discard buttons. For DOCX attachments (Textract doesn't accept them), falls back to `python-docx`; XLSX uses `openpyxl`. Both packages are stable and widely used in 2026, though their maintenance velocity is light — for a price-list path that only runs a few times a month, that's acceptable. If extraction precision becomes a concern, the active community fork `python-docx-oss` is a drop-in alternative. Memory: 512 MB. Timeout: 60 s.

- **`planner`** — S3 event trigger on `menu.json` PUT (plus a daily EventBridge Scheduler audit run that re-diffs every channel to catch drift). Reads `s3://ms-menu-source/menu.json` and the rules and voice docs. For each item-and-channel pair, computes the diff against `ms-state`, checks the change against the auto-sync limits, and decides on a move. Emits one event per pair that needs action: `ms.push`, `ms.hold`, or `ms.flag`, with the item-and-channel context as the event payload. In-sync pairs emit nothing. Memory: 512 MB. Timeout: 60 s. *No Bedrock calls.*
- **`publisher`** — EventBridge rule on the three move events. Resolves the channel list, runs the approval gate (push passes through; hold sends a Slack approval card and waits), formats the change from the voice template, attaches an idempotency key built from `(item_id, channel, value_hash)`, and calls the channel adapter. Writes the result to `ms-state` after the adapter responds. On

a hold, the approval card's Approve button re-invokes `publisher` with the same payload. Memory: 256 MB. Timeout: 30 s.

- **`fix-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on the request body. Triggered by Slack interactive button clicks (Retry/Edit/Skip) on flagged rejections. Writes to `ms-state` and `ms-audit`; on retry, re-invokes the adapter with the same idempotency key; on edit, updates the Drive sheet (or the channel's voice template) via the Sheets API and resends; on skip, marks the pair skipped. Memory: 256 MB. Timeout: 15 s.
- **`digest`** — EventBridge Scheduler target, weekly Sunday 6pm. Reads `ms-state` and `ms-audit` for the past week; sends a digest to a configured Slack channel listing changes pushed, held items awaiting approval, and every channel currently out of step. No Bedrock; the message is a plain summary table. Memory: 256 MB.
- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `ms-state` and `ms-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph owner narrative (how many changes flowed automatically, how many needed a tap, which channel rejects most); emails it via SES. Memory: 512 MB.

Storage

- **DynamoDB · `ms-state`** — one row per item per channel. PK `(item_id, channel)`; attributes: `last_value`, `last_push_status` (pushed/rejected/skipped), `last_push_at`, `reject_reason`,

`idempotency_key`. On-demand. No TTL — this is the live picture of what each channel shows.

- **DynamoDB** · `ms-audit` — one row per write action of any kind. PK `(item_id, ts)`; attributes: `channel`, `action` (push/hold-approved/retry/edit/skip), `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.
- **S3** · `ms-menu-source` — mirrored JSON from the master menu sheet. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3** · `ms-rules-source` — mirrored rules and voice docs as plain text. Versioning enabled.
- **S3** · `ms-raw-mime` — raw inbound MIME from forwarded price lists. Lifecycle to Glacier at 30 days; expiry at 7 years.
- **S3** · `ms-pdf-out` — generated printable menu PDFs, one per regenerate, with the latest under a stable key the website links to.

Channel adapters

An adapter is a small module implementing one contract: `apply(item, formatted_value, idempotency_key) -> {status, reason}` and `read(item) -> current_value`. Each adapter encapsulates one channel's auth, units, and field set.

- `adapter-website` — writes to the site's content store (a headless CMS API or a JSON file in `s3://ms-menu-source/site/` the site reads at build/render). Accepts long descriptions and dollar prices.

- `adapter-platform-*` — one per online-order platform. Calls that platform's menu API (OAuth token in Secrets Manager under `ms/channel/<name>`), converts prices to integer cents, enforces the platform's name/description length caps, and maps categories to the platform's taxonomy. These are the adapters most likely to return a rejection.
- `adapter-pdf` — regenerates the printable menu (a small HTML-to-PDF render in Lambda) and writes it to `s3://ms-pdf-out/`. Always accepts; the "rejection" case here is a render error, which is flagged the same way.
- `adapter-qr` — updates the QR-code landing page (the same JSON the website reads, or a dedicated key). Accepts the full field set.

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-ses-parser` for the supplier-price matching, and `summary` for the monthly owner narrative. The heavier `anthropic.claude-sonnet-4-6` profile is wired but unused at SMB volume; it's the upgrade path only if price-list matching across a very large menu needs deeper reasoning.
- **Embeddings.** Not used. The menu is structured rows; deterministic comparison beats vector retrieval here. No Knowledge Base, no S3 Vectors. (If supplier lists ever need fuzzy item matching at scale, Amazon Titan Text Embeddings V2 at 1024-dim over S3 Vectors is the path — not needed today.)
- **Quotas.** Default account quotas are more than enough at SMB volume. The planner doesn't call Bedrock; the parsing lane fires a few times a month at

most.

EventBridge config

- `ms-move-rule` — rule on the default bus matching `ms.push`, `ms.hold`, `ms.flag`. Target: `publisher` Lambda.
- `ms-menu-sync-poll` — EventBridge Scheduler, `rate(5 minutes)`, fallback if a Drive push notification is missed. Target: `menu-sync` Lambda.
- `ms-daily-audit` — `cron(0 4 * * ? *)` in TZ. Target: `planner` Lambda, forced full re-diff of every channel to catch silent drift.
- `ms-weekly-digest` — `cron(0 18 ? * SUN *)` in TZ. Target: `digest` Lambda.
- `ms-monthly-summary` — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `prices.your-restaurant.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `ms-inbound-rules`: one rule with recipient `prices@your-restaurant.com` → spam scan → S3 PUT to `s3://ms-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses-parser`.
- SES outbound for the digest and summary emails: verify a sender identity at `menu@your-restaurant.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:

- **planner role:** `s3:GetObject` on the menu, rules, and voice keys; `dynamodb:Query` + `GetItem` on `ms-state`; `events:PutEvents` on the default bus. *No* `bedrock:*`.
- **publisher role:** `secretsmanager:GetSecretValue` on the channel and Slack secrets; `dynamodb:PutItem` on `ms-state`; `s3:PutObject` on `ms-pdf-out` and the site key; outbound network access to `hooks.slack.com` and each channel's API host.
- **fix-handler role:** `dynamodb:PutItem` on `ms-state` and `ms-audit`; `secretsmanager:GetSecretValue` on the Sheets-API and channel secrets; outbound network access to `sheets.googleapis.com` and channel hosts; `dynamodb:Query` for state lookup.
- **intake-ses-parser role:** `s3:GetObject` on `ms-raw-mime`; `textract:StartDocumentTextDetection` + `StartDocumentAnalysis`; `bedrock:InvokeModel` on the Haiku ARN; `secretsmanager:GetSecretValue` on the Slack webhook.
- **menu-sync and quick-edit roles:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on the menu 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 approval and flag messages 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 `fix-handler` Function URL (held approvals route to the `publisher` via the same handler). `fix-handler` verifies the Slack signing secret on the inbound request, parses the action_id (`retry`, `edit`, `skip`, `approve`, `reject`), opens a modal if needed (Edit opens a modal; Retry/Skip/Approve are 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 `ms/slack/bot-token`. The signing secret is `ms/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:** publisher rejection rate > 5% in 24h (a channel might have changed its API); planner failures > 0 in a day; fix-handler signature-verification failures > 5/hour (might mean the Slack secret rotated).
- **X-Ray:** off by default. Not worth the cost at SMB volume.
- **AWS Budgets:** $25/month threshold, alarm at 80% and 100%, posts to SNS topic `ms-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Service-account credentials for Drive and Sheets APIs live in Secrets Manager under `ms/drive/sa`. Each channel's API token lives under `ms/channel/<name>`. Slack bot token, signing secret, and webhook URL all under `ms/slack/*`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, auto-sync limits, approval thresholds, staff PIN, and channel list all live in Parameter Store under `/ms/config/`. Lambdas fetch config on cold start and cache for the lifetime of the execution environment.

Deploy

GitHub Actions + OIDC + AWS SAM, no long-lived keys — the CI role is assumed via an OIDC trust policy scoped to the repo. The opinionated bits: deploy the SES rule set as a separate stack (rule-set changes affect mail flow), turn on S3 versioning for both `ms-menu-source` and `ms-rules-source` so a bad Drive edit can be rolled back in one click, and keep each channel adapter behind a feature flag so a misbehaving platform can be paused without a redeploy. Total deployable surface: around nine Lambdas, two 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.