7-PART SERIES · FREE COMPANION

Proposal generator

A serverless generator that turns a short brief — client, what they need, rough budget — into a polished first-draft sales proposal: a cover, an understanding of the need, a proposed approach, a timeline, and a price summary, pulled from your own templates and past winning proposals, in your voice. You edit and send; nothing reaches a client without sign-off. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Proposal generator

PART 1 OF 7

JUNE 2, 2026 PART 1 OF 7 · PROPOSAL GENERATOR SERIES ~5 MIN READ

A proposal generator on AWS for a few dollars a month

A small business wins work on proposals, and the good proposals take hours nobody has. A prospect calls on Tuesday, loves the chat, and asks you to “send something over.” By Friday the thread has gone cold because the blank page won. You already know what you’d write — the cover, the bit where you show you understood the need, the approach, the timeline, the price — because you’ve written it forty times before. This post walks through the design of a small system that takes a few notes from you, drafts the whole proposal in your voice from your own past work, and hands it back for you to edit and send. Nothing goes to a client without your sign-off.

KEY TAKEAWAYS

- Three ways to start a proposal: a short web form, a forwarded email thread, or a one-line Slack command.
- Every draft has the same five parts: cover, understanding of the need, approach, timeline, and price summary.
- The draft is grounded on your own templates and the past proposals that won — so it sounds like you.
- The price is computed in plain Python from your rate card. The model never invents a number.
- Designed on AWS for about $3/month at typical small-business volume.

The whole system on one page

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

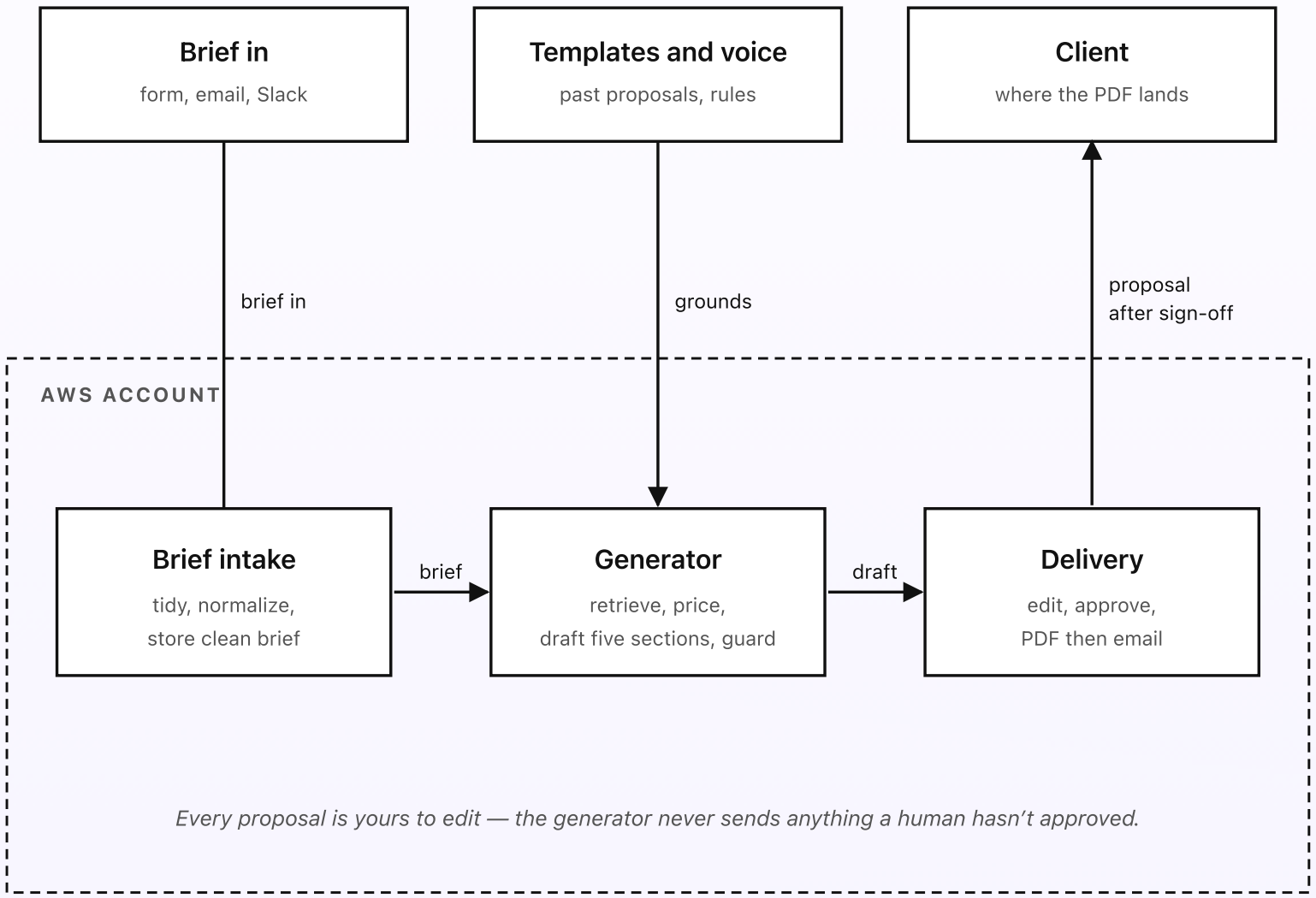

Brief in
form, email, Slack
Templates and voice
past proposals, rules
Client
where the PDF lands
brief in
grounds
proposal
after sign-off
AWS ACCOUNT
Brief intake
tidy, normalize,
store clean brief
brief
Generator
retrieve, price,
draft five sections, guard
draft
Delivery
edit, approve,
PDF then email
Every proposal is yours to edit — the generator never sends anything a human hasn't approved.

Fig 1. Three sources outside, three pieces inside AWS. A brief flows in from a form, an email, or a Slack command. The Generator drafts five sections grounded on your past work. Delivery shows you the draft, then sends the approved proposal as a PDF.

What you set up once (the outside)

- **Brief in.** Three ways to start a proposal, covered in Part 2. A short web form (client name, what they need, rough budget, due date) for when you're at your desk. A forwarded email thread — send the back-and-forth with the prospect to a dedicated address and the system reads it into a brief. A one-line Slack command for the proposal you remember on the way out the door. All three end as the same clean brief.
- **A templates folder.** A Google Drive folder you fill once. One sub-folder holds your *section templates* — the cover format you like, a scope shell, a standard timeline, your terms. One holds your *past proposals*, the ones that won; the generator reads these to sound like you. One short doc holds your *pricing rules and voice notes* — your day rate, your packages, the phrases you always use and the claims you never make. Editing any of these changes future drafts without touching code.
- **Client.** Where the finished proposal lands. After you approve a draft, the system renders it to a branded PDF, saves a copy to the Drive folder, and emails it to the client with a short cover note. You can also just download the PDF and send it yourself.

What runs on every brief (the inside)

- **The brief intake.** Whichever lane the brief came in on, a small step tidies it: a cheap Bedrock Haiku 4.5 call pulls out the client name, the need, the rough

budget, and the due date, and flags anything missing. The clean brief is stored, and you get a one-tap “start the draft” card. Nothing is drafted until you say go — the intake just gets the brief ready.

- **The generator.** The heart of the system, covered in Part 3. It finds your closest past proposals (using embeddings — a way to find work that reads *similar* to this brief, not just work that shares the same words). It computes the price in plain Python from your rate card and the brief’s budget. Then it makes one grounded Claude Sonnet 4.6 call to write all five sections — cover, understanding of the need, approach, timeline, and price summary — using your templates, the retrieved past sections, and the computed numbers. A guard re-reads the draft to check every price and date matches the computed values before anyone sees it.
- **Delivery.** The draft shows up for you to read — in Slack with three buttons, or in a linked Google Doc you can edit directly. *Approve* renders a branded PDF and emails it to the client. *Edit* opens the doc to revise, then re-approve. *Discard* archives the brief. Every action is logged so a sent proposal can always be traced back to the brief it came from. A weekly digest lists what went out and what’s still sitting in draft.

In plain words

A prospect named Delgado & Co. asks you to redesign their booking flow. You type three lines into the form: “Delgado & Co. — rebuild the online booking flow, mobile-first — budget around $18k — wants it live before their Q4 push.” You tap *start the draft*. Ninety seconds later a draft lands in your Slack: a cover addressed to Delgado & Co., a paragraph that plays back exactly what they need, an

approach lifted from the two booking projects you've done before, a four-week timeline, and a price summary that lands at $17,500 against their budget — computed from your day rate, not guessed. You tweak two sentences in the linked doc, change the timeline by a week, and tap *Approve*. The branded PDF emails itself to Delgado & Co. with your cover note. Total time: under ten minutes, on a Tuesday, before the thread went cold.

The cost of running this is about $3 a month at SMB volume. The cost of *not* running it is the proposal you didn't send because Friday got busy — the deal that quietly went to whoever did get something over.

DESIGN RULES THAT SHAPED EVERY DECISION

- Five sections, always. Cover, understanding of the need, approach, timeline, price summary. There is no sixth.
- The price is computed in code from your rate card. The model restates the number; it never invents it.
- Every draft is grounded on your own templates and past winning proposals — so it reads like you, not like a robot.
- Nothing is sent without a human tapping Approve. The default action is "show me," never "send."
- The templates live in Drive. Changing your voice, packages, or terms doesn't need a deploy.
- Every proposal is logged. Look at a deal next quarter and you can see the brief it came from and who approved it.

Why this shape

Most small teams write proposals one of two ways: from a blank page every time, or by copying last month's proposal and find-and-replacing the client name. The blank page is slow and it's where deals go to die — the proposal you mean to write tonight becomes the one you never send. The copy-paste approach is fast but dangerous: it's how last client's name ends up in this client's cover, or how a price from a different-sized job sneaks in.

The setup above keeps the best of both. The draft is fast, because the system writes it. It sounds like you, because it's grounded on your own winning proposals instead of a generic template. And it's safe, because the numbers are computed in code and a human approves every word before it leaves. The generator does the eighty percent that's the same every time, and leaves you the twenty percent — the judgment, the tweak, the "actually let's phase this" — that's why a person should still be in the loop.

The next four posts walk through each piece in turn: how a brief gets captured, how a proposal gets drafted, how a proposal stays on brand, and how a proposal gets sent. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

JUNE 2, 2026 PART 2 OF 7 · PROPOSAL GENERATOR SERIES ~4 MIN READ

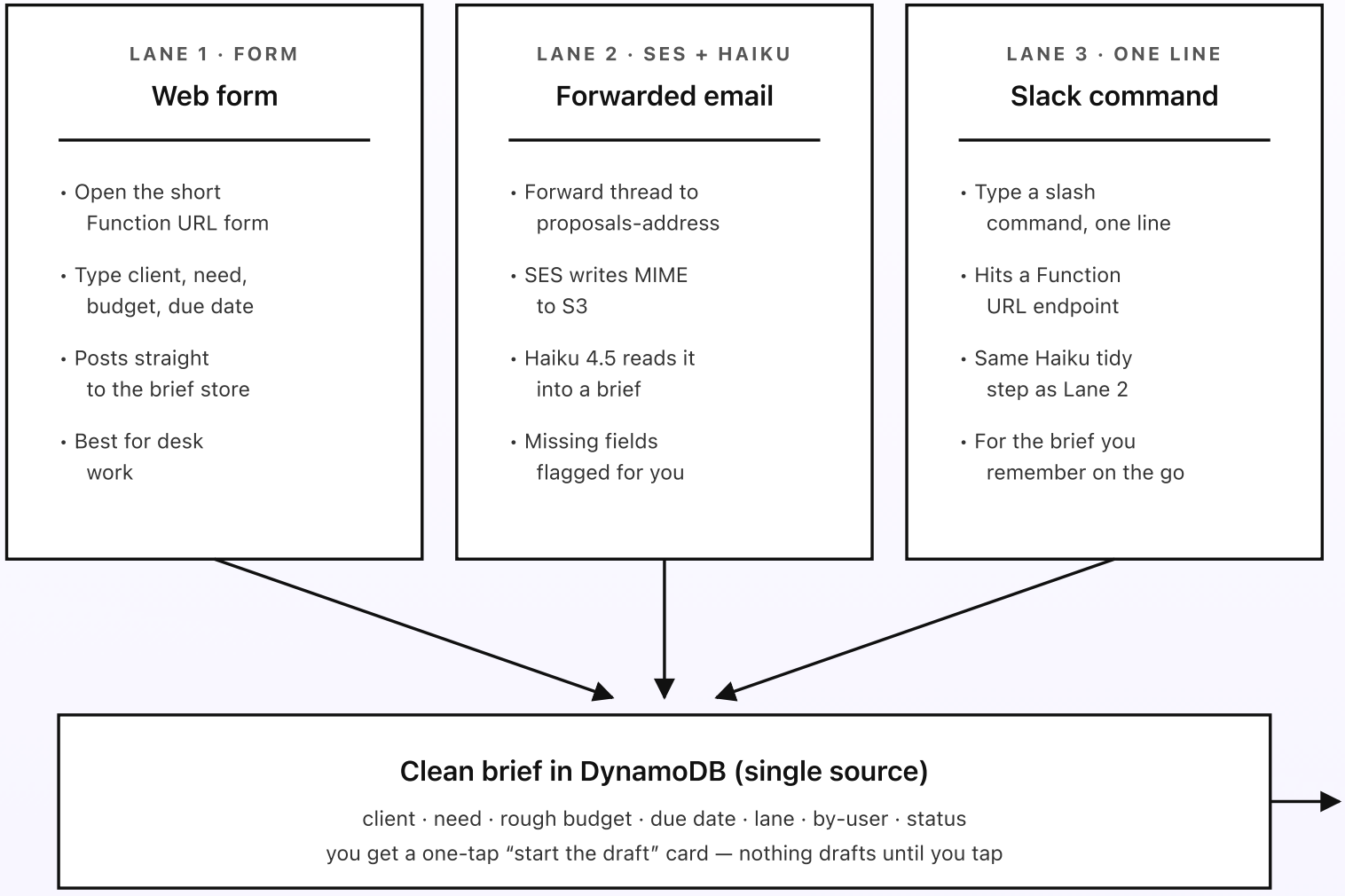
How a proposal brief gets captured

The generator only drafts what it's told. So the first job is making it dead easy to tell it — because the proposal you never start is the one you lose. There are three ways a brief gets in: you fill a short web form, you forward the email thread with the prospect, or you type a one-line command in Slack. The form is the obvious one. The other two exist because in real life the moment you decide to send a proposal is rarely the moment you're sitting at a form.

KEY TAKEAWAYS

- Three intake lanes feed one clean brief: a web form, a forwarded email thread, and a Slack command.
- Forwarded threads are read by Bedrock Haiku 4.5 into a brief — client, need, budget, due date.
- Every brief is tidied into the same shape, and anything missing is flagged before you start.
- You always get a one-tap “start the draft” card. Nothing drafts until you tap it.
- The clean brief is the one thing the generator reads. The lanes are just three doors into it.

Three lanes into one brief



The clean brief is the single source — the three lanes are just three doors into it.

Fig 2. Three lanes converge on one clean brief. The web form posts straight in; the email and Slack lanes are read by a cheap model call into the same shape. You always confirm with a one-tap card before any drafting starts.

Lane 1: the web form

The simplest lane. A short form, served by a Lambda Function URL (a plain web address that runs a small function when you submit — no heavyweight web gateway behind it). Four fields: client name, what they need (a sentence or two is fine), a rough budget, and a due date. You hit submit and the form writes a brief straight into the `pg-briefs` table in DynamoDB. No model call is needed here — you already typed clean fields.

This lane is for when you're at your desk and you know exactly what you want. Most planned proposals start this way. The form is the one place the system asks for structure up front; the other two lanes do that work for you.

Lane 2: forwarded email (the lane most people actually use)

Set up a dedicated address — something like `proposals@your-company.com` — through Amazon SES. When a prospect emails you and you decide it's worth a proposal, you forward the whole thread to that address and the system takes it from there. SES writes the raw message to `s3://pg-raw-mime/`. That write triggers a parser Lambda. The Lambda strips the email down to plain text (quoted replies, signatures, and disclaimers removed) and calls Bedrock Haiku 4.5 to read a brief out of the conversation.

The model's job here is small and bounded: "Read this email thread. Return a brief as JSON: client name, what they need, any budget mentioned, any deadline

mentioned. If a field isn't in the text, leave it blank — do not guess." The result becomes a clean brief with a status of "needs review." You get a Slack card showing what it read, with any blank fields highlighted, and a button to start the draft or fix the brief first. The thread itself is kept in S3 so the generator can quote a detail from it later if it helps.

Why does a human still confirm? Because a brief the model misread is cheap to fix now and expensive to fix after it's shaped a whole proposal. Thirty seconds of "yes, that's right" beats rewriting a draft built on the wrong budget.

Lane 3: the Slack command

Sometimes you remember the proposal you owe someone while you're between meetings. Lane 3 is for that. A slash command in Slack — `/proposal Delgado & Co., rebuild booking flow, ~$18k, before Q4` — hits a Function URL. The same Haiku tidy step from Lane 2 turns your one line into a structured brief: client, need, budget, due date. You get the same one-tap card back in the same channel.

The Slack lane trades completeness for speed. You won't always have the budget or the deadline in your one line, and that's fine — the missing-field flag tells you what the draft will have to assume, and you can fill it in before or after the first draft. It exists so that "I should send that proposal" turns into an actual brief in the ten seconds you have, instead of a note-to-self that never happens.

Why everything funnels to one clean brief

Three lanes in, but only one thing the generator reads: the clean brief in DynamoDB. That's deliberate. If each lane handed the generator a slightly

different shape, every “why did the draft say that?” question would mean checking which door the brief came through. Funneling everything into one brief shape means there is exactly one record per proposal, with the same fields every time, no matter how it started. The lanes are conveniences for getting a brief in; they all pass through the same clean brief on the way.

Next post: how the generator takes that clean brief, finds your closest past proposals, computes the price, and writes all five sections in one grounded call.

PART 3 OF 7

JUNE 2, 2026 PART 3 OF 7 · PROPOSAL GENERATOR SERIES ~5 MIN READ

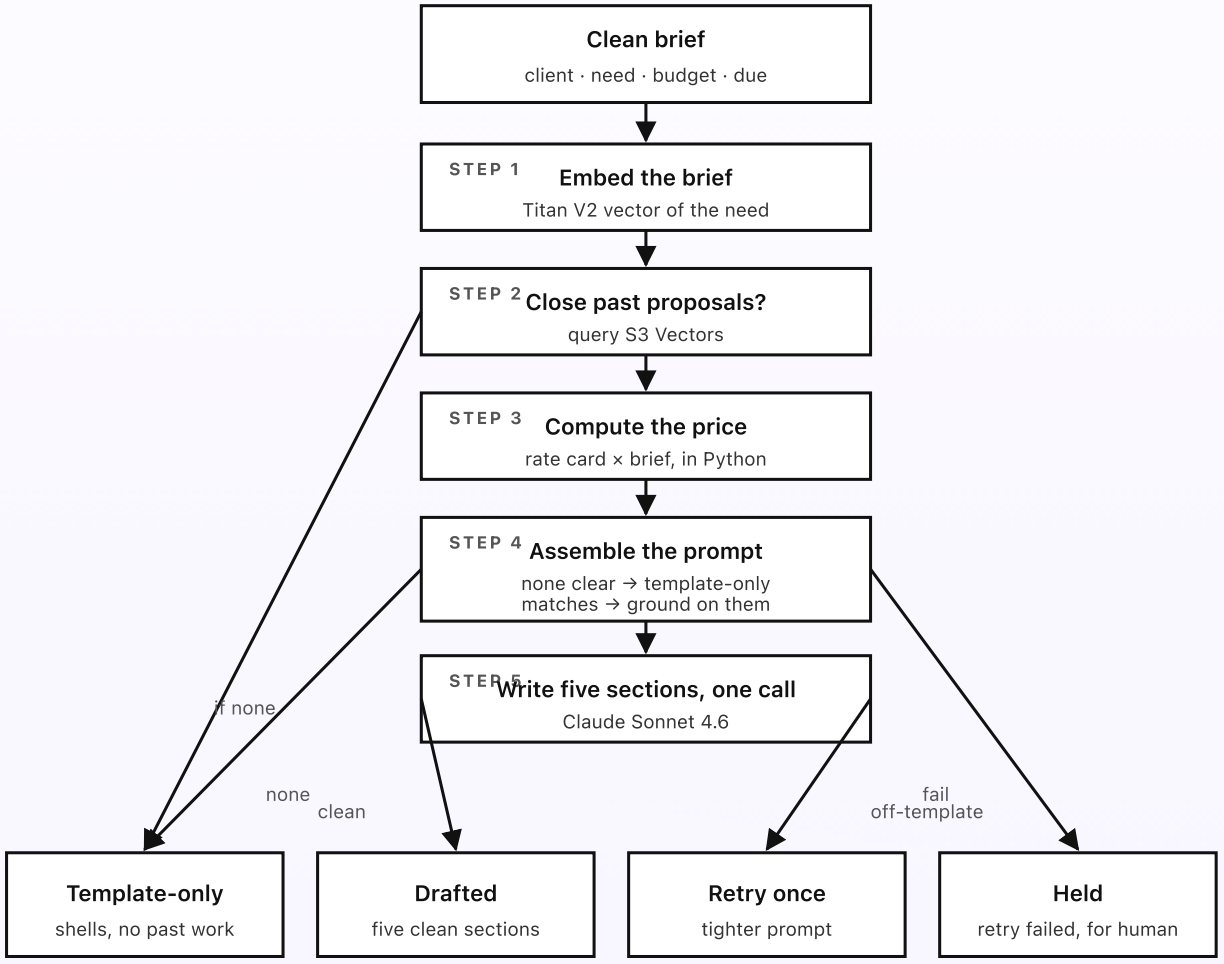
How a proposal gets drafted

You tapped “start the draft.” Now the generator has about ninety seconds to turn three lines of brief into a proposal that sounds like you wrote it. It does that in five steps: read the brief, find your closest past work, compute the price in plain code, write all five sections in one grounded call, then check the draft before anyone sees it. The model writes the words. It never decides the numbers.

KEY TAKEAWAYS

- The build job runs as one short Lambda, kicked off the moment you tap “start the draft.”
- Retrieval finds your closest past proposals using Titan embeddings over S3 Vectors.
- The price summary is computed in plain Python from your rate card — no model touches the math.
- One Claude Sonnet 4.6 call writes all five sections, grounded on your templates and retrieved work.
- A guard re-reads the draft so the prose can only repeat numbers that were computed.

The build flow, per brief



The rate card holds every number — change a rate and the next draft prices itself.

Fig 3. The build flow, per brief, when you start a draft. Five steps turn a clean brief into a draft. Retrieval grounds it on your past work; the price is computed in code; the model only writes the prose.

Steps 1–2: find your closest past work

Your past proposals are your best template — better than any generic shell, because they already sound like you and they already solved problems like this one. The trick is finding the *right* past proposals for this brief. The system does that with embeddings: when each past proposal is added to the Drive folder, the generator turns each of its sections into a vector — a string of numbers that captures what the text is about — using Amazon Titan Text Embeddings V2, and stores it in S3 Vectors (Amazon's built-in way to search vectors without running a separate database).

When a new brief comes in, the generator embeds the need the same way and asks S3 Vectors for the nearest past sections. A brief about "rebuild the booking flow" pulls up the approach and timeline from your two past booking jobs, even if those proposals never used the word "rebuild." If nothing clears a similarity bar — say it's the first proposal of a kind you've never done — the generator falls back to the plain section templates and notes that to you, so you read the first draft a little more carefully.

Step 3: compute the price in plain Python

This is the most important rule in the whole system: the model does not decide the price. A small Python function reads your rate card and packages from the

rules doc, looks at the brief's scope and rough budget, and computes a price summary — line items, subtotal, any discount, total — as plain numbers. If the brief says "around $18k" and your day rate and the scope land at $17,500, that's the number; the function doesn't round up to match the budget or invent a figure to fill a gap.

Keeping the math in code, not the model, is what makes the price trustworthy. A model asked to "write a price summary" will happily produce a confident, wrong number. A Python function with your rate card produces the same number every time, and you can read the function. The model is handed the computed total and told to present it — not to do arithmetic.

Steps 4–5: assemble, then write all five sections in one call

Now the generator builds one prompt. Into it go: the clean brief, the retrieved past sections, the plain section templates, the voice notes from the rules doc, and the computed price summary. The instruction is specific: "Write a sales proposal with exactly these five sections — cover, understanding of the need, approach, timeline, price summary. Match the voice of the examples. Use the price summary exactly as given; do not change any number. Do not promise anything the templates don't support."

One Claude Sonnet 4.6 call returns all five sections as structured fields. Sonnet 4.6 is the heavier model in the stack, used here because writing a whole coherent proposal — one that reads as a single voice across five sections and stays faithful to the grounding — is real reasoning work, the kind that earns the cost. The

cheaper Haiku 4.5 handles the small jobs (tidying the brief, classifying scope); the draft is where the better model pays for itself. If the model omits a section or wanders off the templates, the generator retries once with a tighter prompt; if the retry also fails, the brief is held for a human rather than shipping a broken draft.

Why one call instead of five? Coherence. A proposal written section-by-section in separate calls reads like five people wrote it — the approach promises something the timeline doesn't deliver, the cover oversells what the scope under-delivers. Writing all five in one pass, with the whole brief in view, keeps the document consistent with itself.

Why split the work this way

The pattern here is the same one that runs through the whole series: do the deterministic parts in code, and let the model do only the part that genuinely needs language. Retrieval is a search problem — code. Pricing is arithmetic — code. Deciding whether a draft is structurally complete — code. The one job left for the model is the one it's actually good at: writing five sections of fluent, on-voice prose from grounded material. By the time Sonnet runs, every fact it needs has already been decided; its job is to say those facts well.

Next post: how the draft stays on brand — the voice rules, the banned-claim checks, and the price-and-date guard that re-reads the draft before it ever reaches you.

PART 4 OF 7

JUNE 2, 2026 PART 4 OF 7 · PROPOSAL GENERATOR SERIES ~5 MIN READ

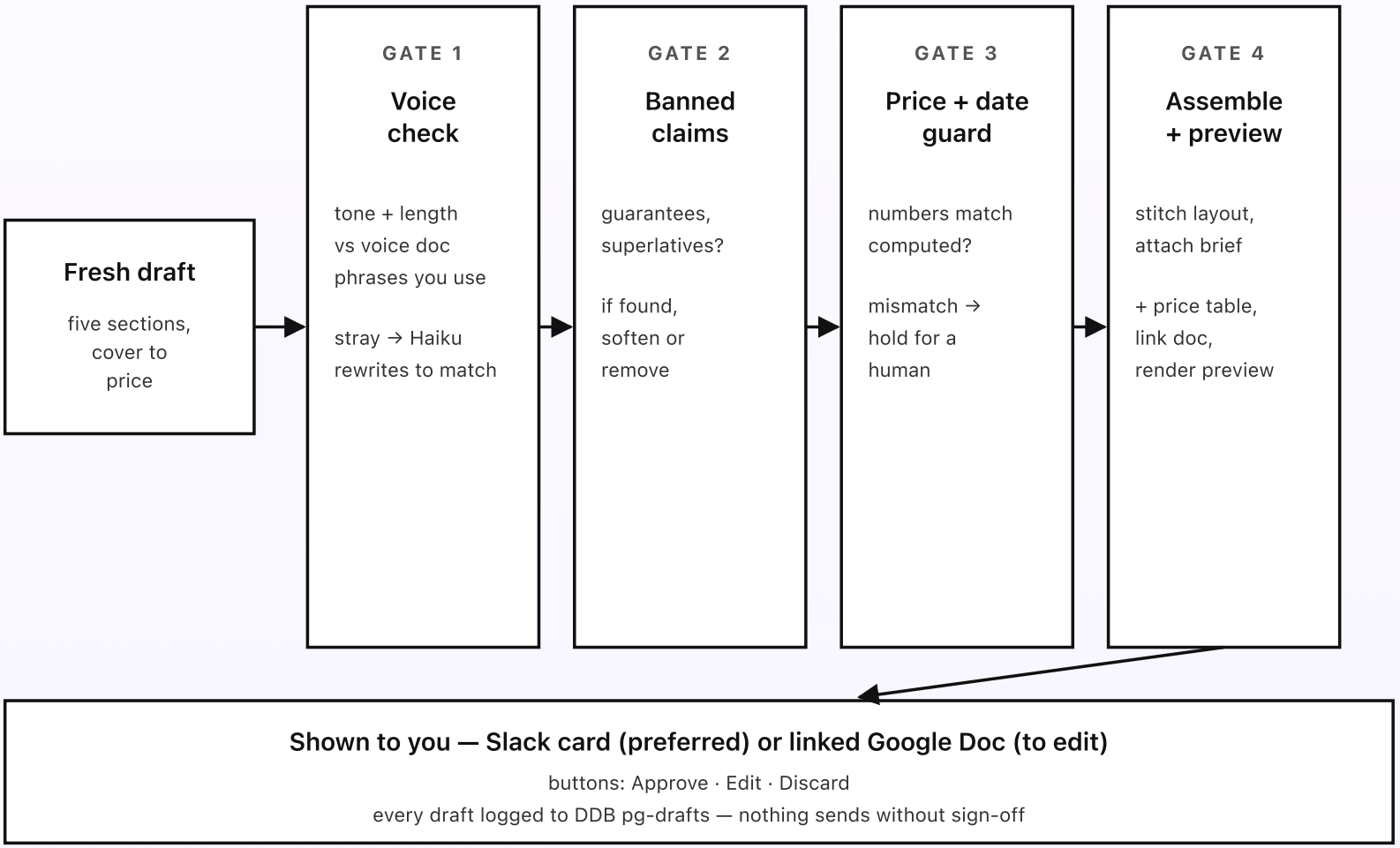
How a proposal stays on brand

The generator wrote a draft. Before it reaches you — let alone a client — it passes through four checks. Each one catches a different way a draft can embarrass you: the wrong tone, a promise you'd never make, a price or date that drifted from the real number, a missing piece. Get any of these wrong and a confident-looking proposal does real damage. Four small gates sit between a fresh draft and your sign-off.

KEY TAKEAWAYS

- Gate 1 checks the draft against your voice rules: tone, length, the phrases you do and don’t use.
- Gate 2 scans for banned claims — guarantees, superlatives, anything your templates don’t support.
- Gate 3 re-reads every price and date and compares them to the computed numbers; any mismatch holds the draft.
- Gate 4 assembles the proposal, attaches the brief, and renders a preview for you.
- Every gate is a check, not a sender. The draft always stops at you for sign-off.

Four gates before sign-off



Every gate is a check, not a sender — the draft always stops at you for sign-off.

Fig 4. Four gates between the fresh draft and your sign-off. Check the voice. Strip banned claims. Guard every price and date. Assemble and preview. Then show it to you — never to the client — with Approve, Edit, and Discard.

Gate 1: the voice check

A proposal that's correct but sounds like a stranger still loses. Gate 1 compares the draft against the voice notes in your rules doc: how long your sentences run, how formal you are, the phrases you reach for ("here's how we'd approach it") and the ones you avoid ("synergize," "cutting-edge"). Most of this check is plain pattern-matching in code — a banned-phrase list, a sentence-length range, a reading-level target.

If the tone strays far enough, a short Haiku 4.5 pass rewrites the offending section to match your voice, then the check runs again. Haiku is the cheap model, and rewriting a paragraph to a known style is exactly the kind of small, well-defined job it's good at. The expensive Sonnet call already happened; this is a light touch-up, not a re-draft.

Gate 2: banned claims

The fastest way to get a proposal into legal trouble — or just to sound like everyone else — is an unearned promise. "Guaranteed results." "The best in the industry." "100% uptime." Gate 2 scans the draft for a configurable list of banned claims: guarantees, superlatives, absolute numbers you can't back, and any promise the section templates don't explicitly support. A flagged claim is softened ("guaranteed" becomes "we aim for") or removed.

The banned list lives in the rules doc, so you can tune it without a deploy. A consultancy and a builder will draw the line in different places; the system enforces wherever you draw it. This gate is the difference between a draft that's confident and one that's reckless.

Gate 3: the price-and-date guard

This is the gate that matters most, and it's pure code. Part 3 computed the price summary and the timeline dates in plain Python. Gate 3 re-reads the finished prose, pulls out every number and every date the model wrote, and compares each one against the computed values. The total in the cover has to match the total in the price table. The "four weeks" in the approach has to match the dates in the timeline. A figure in the prose that doesn't match a computed value is a red flag — the model either restated a number wrong or invented one.

Any mismatch holds the draft. It doesn't quietly fix the number (a quiet fix could mask a real bug), and it never ships the wrong figure to you hoping you'll catch it. It stops, flags exactly which number disagreed with which computed value, and routes the draft to a human. In steady state this gate almost never fires — the model is told to use the numbers verbatim — but it's the backstop that lets you trust the price on every draft you *do* see.

Gate 4: assemble and preview

The last gate isn't a filter; it's the packaging. It stitches the five approved sections into your proposal layout, attaches the original brief and the computed price table, creates a Google Doc copy you can edit directly, and renders a preview. Then it

shows you the result — as a Slack card with Approve, Edit, and Discard buttons, or as the linked doc.

Note what Gate 4 does *not* do: send anything. It assembles and previews. The proposal sits in a "ready for review" state, logged to the `pg-drafts` table, until you act on it. The next post is about what those three buttons actually do.

Why the guardrails exist

None of these gates are exotic. They're the care a careful person takes before sending a proposal under their own name: does this sound like us, did we promise anything we shouldn't, are the numbers right, is it actually finished. Putting them in code as four sequential gates makes that care automatic instead of something you're trusting yourself to remember at 6pm on a Friday. The model is fast and fluent; the gates are what make it safe to be fast.

Next post: how a proposal gets sent — the three actions on the Approve button, and how the registry of sent proposals stays in sync with the audit trail.

PART 5 OF 7

JUNE 2, 2026 PART 5 OF 7 · PROPOSAL GENERATOR SERIES ~5 MIN READ

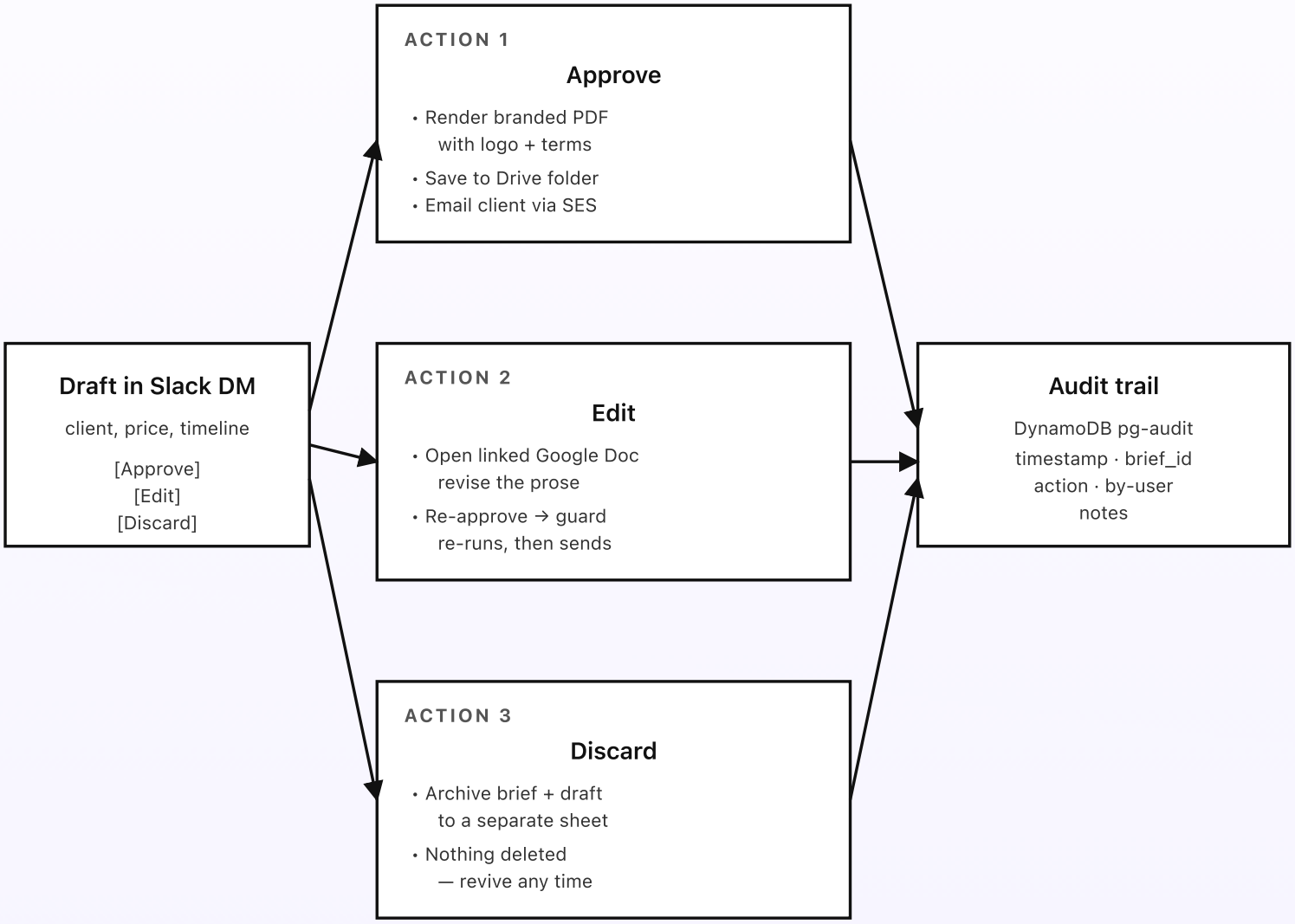
How a proposal gets sent

A draft lands in your Slack at 2:14pm. The cover reads right, the price is $17,500, the timeline is four weeks. There's an Approve button. What happens when you tap it? The honest answer is "it depends on what you actually did." This post walks through the three things you can do with a finished draft — approve, edit, discard — and how the PDF, the Drive folder, and the audit trail all stay in sync.

KEY TAKEAWAYS

- Three actions per draft: *approve* (render and email), *edit* (revise in the doc), *discard* (archive).
- Approve renders a branded PDF, saves it to Drive, and emails the client via SES with a short cover note.
- Edit opens the linked Google Doc; your changes flow back and you re-approve the revised draft.
- Discard archives the brief and the draft — nothing is deleted, so a dropped deal can be revived.
- The Approve button is a Slack interactive message backed by a Function URL.

Three actions on Approve



Discard doesn't delete — it archives. A dropped proposal can always be reopened from the brief.

Fig 5. Three actions per draft, three different effects. Approve renders and emails the proposal. Edit opens the doc to revise, then re-runs the guard and sends. Discard archives without deleting. Every action writes to the audit trail.

Action 1: approve (the most common)

You read the draft, it's right, you tap *Approve*. The button submits to a Function URL Lambda, and four things happen in order. First, the five sections are rendered into a branded PDF — your logo, your colors, your standard terms page, the computed price table laid out cleanly. Second, the PDF is saved to the `Proposals/sent/` folder in Drive, named with the client and date so it's easy to find later. Third, the PDF is emailed to the client through SES outbound, with a short cover note pulled from your voice doc ("Hi — great speaking earlier. Attached is our proposal; happy to walk through it."). Fourth, an `action: sent` row is written to `pg-audit` with you, the timestamp, the brief id, and the final price.

From the client's side, this looks like you sat down and wrote them a thoughtful proposal that afternoon. From your side, it was a ten-minute review and one tap. The draft that would have died on the blank page is in their inbox before the conversation cooled.

Action 2: edit (the tweak)

Most drafts need a small change. The approach is right but you'd phase it differently. The timeline assumed a start date that slipped. A sentence in the cover is almost right. *Edit* opens the linked Google Doc — the same doc Gate 4 created

— and you revise the prose directly, in a tool you already know, with no new app to learn.

When you're done, you re-approve from the doc or back in Slack. The Lambda pulls the edited text back, and — this is the important part — runs the price-and-date guard again. If your edit changed a number, the guard re-checks it against the computed values; if you deliberately overrode the price, the guard records that as a human override in the audit trail rather than blocking you. Then it renders and sends exactly as *Approve* would. An edit never skips the guard; it just lets you be the one who decides.

Action 3: discard (the "not this one")

Sometimes the draft isn't worth sending. The lead went quiet. You decided to pass. The scope was wrong and it's easier to start fresh than to fix. *Discard* archives the brief and the draft to a separate *archived* sheet in the Drive folder, with the date and who discarded it. Nothing is deleted.

That "nothing is deleted" matters. Leads come back. Three weeks later the prospect emails "sorry, got pulled away — still keen?" and you can reopen the archived brief, re-run the draft against your current rate card, and send in minutes. A discarded proposal is paused, not lost. The archived sheet is also what the monthly summary in Part 6 reports on — how many briefs came in, how many went out, how many got dropped and why.

Every action is logged, every proposal is traceable

The `pg-audit` table records every approve, edit, and discard with the user who acted, the timestamp, and a snapshot of the draft before and after. Months later, when a client signs and someone asks “what exactly did we promise them?”, the answer is one lookup: the brief it came from, the version that was sent, the price that was quoted, and who approved it. If a wrong proposal somehow goes out, the snapshot lets you see exactly what was sent and send a correction with a clean paper trail.

This traceability is the quiet payoff of keeping a human in the loop and logging every step. The generator makes you fast; the audit trail makes you accountable. Neither is worth much without the other.

Next post: the cost breakdown. The whole pipeline above runs in coffee-money territory at SMB volume; Part 6 explains exactly where the dollars go and why the one Sonnet draft call is the bill.

PART 6 OF 7

JUNE 2, 2026 PART 6 OF 7 · PROPOSAL GENERATOR SERIES ~3 MIN READ

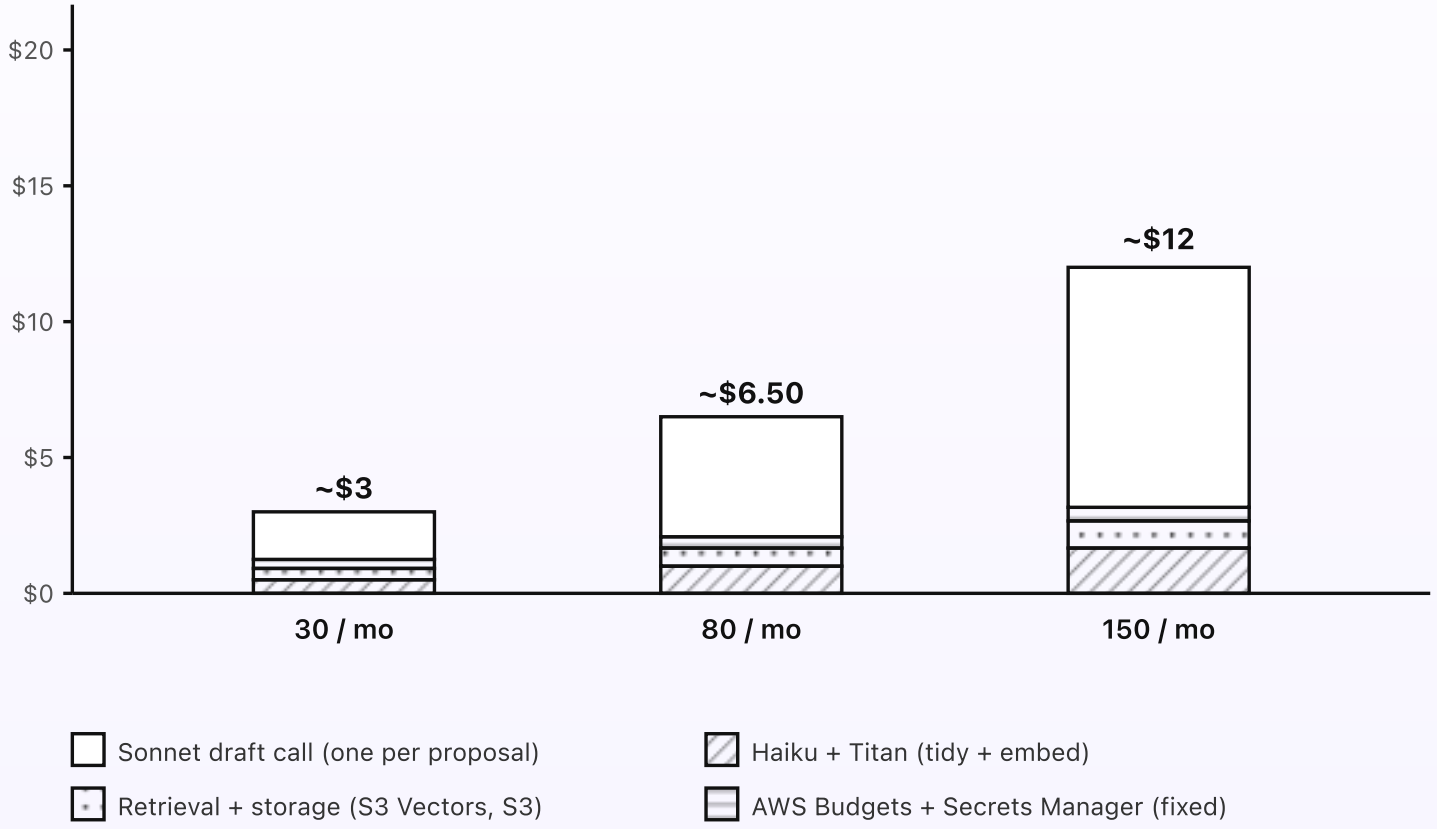
What the proposal generator costs

The generator only does real work when you start a draft. It reads a brief, searches your past proposals, runs some Python for the price, makes one bigger model call to write the proposal, and renders a PDF. Everything else is idle. The one Claude Sonnet 4.6 draft call is the bill; the cheap tidy and search steps, the storage, and the build Lambdas are slivers. At typical SMB volume, that's a few dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $3/month at typical SMB volume (about 30 proposals a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- One Sonnet 4.6 draft call per proposal is the dominant cost — cents per proposal.
- The cheap Haiku tidy, Titan embeddings, and S3 Vectors retrieval are small slivers.
- At 80 proposals a month the bill is around $6.50. At 150 it's around $12.

Cost at three volumes



The draft call is the dominant cost — and even that is a few cents per proposal.

Fig 6. Monthly cost at three proposal volumes. The Sonnet draft call dominates because one is written per proposal. The cheap steps, retrieval, and storage are small slivers that barely grow.

Where the dollars actually go

Bedrock Sonnet draft (the bulk). One Claude Sonnet 4.6 call writes the whole proposal — five sections — per draft. That's a few thousand input tokens (the brief, the retrieved past sections, the templates) and a couple of thousand output tokens (the proposal prose). At Sonnet pricing that's a few cents per proposal. Thirty proposals a month is a couple of dollars; this single call is most of the bill at every volume.

Bedrock Haiku + Titan (the cheap steps). The brief tidy, the voice touch-ups in Gate 1, and the scope classify all run on Haiku 4.5 — a fraction of a cent each. Titan Text Embeddings V2 embeds each brief and each new past proposal — also a fraction of a cent. Together these are a small sliver next to the Sonnet draft.

Retrieval + storage. S3 Vectors holds the embeddings of your past proposal sections and answers the nearest-match query during each draft — pennies a month at this scale. S3 holds the past proposals, the rendered PDFs, and the forwarded email threads — a few MB total, effectively free.

Lambda runtime. The build job, the brief intake, the drive-sync every fifteen minutes, the render-and-send Lambda, the ack-handler. None run for long. The whole Lambda total lands well under a dollar at all three volumes.

DynamoDB on-demand. Three small tables: `pg-briefs`, `pg-drafts`, `pg-audit`. A handful of reads and writes per proposal. Pennies a month at any of these volumes.

SES. Inbound for the forwarding lane and outbound for the approved proposals: $0.10 per thousand messages either way. A few cents a year for an SMB.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the brief form and the approve button.
- **NAT Gateway.** Nothing is in a VPC. No NAT, no $32/month minimum.
- **Always-on compute.** No EC2, no Fargate. The generator only runs when you start a draft.
- **A separate vector database.** S3 Vectors handles retrieval without a running cluster to pay for.
- **A model on the price.** The price summary is plain Python. No model call, and no risk of a made-up number.

How the cost scales

The Sonnet draft call grows linearly with proposal count, because one draft is written per proposal. Everything else — storage, the cheap steps, the build Lambdas — barely moves. So the bill at 300 proposals a month is around $24; at 600 it's around $45. Past those volumes you're writing twenty proposals a day, which is a great problem to have, and the only optimization worth making is caching the retrieved sections for repeat client types — an optimization, not a redesign.

Set an AWS Budgets alarm at $15/month so anything unusual pages you before the bill matters. At typical SMB volume the generator's bill stays comfortably under that ceiling.

Last post in the series: the engineering reference. Same system, drawn for engineers — service names, Lambda inventory, IAM scopes, the S3 Vectors setup, DynamoDB schemas, and EventBridge config.

PART 7 OF 7

JUNE 2, 2026 PART 7 OF 7 · PROPOSAL GENERATOR SERIES ~8 MIN READ

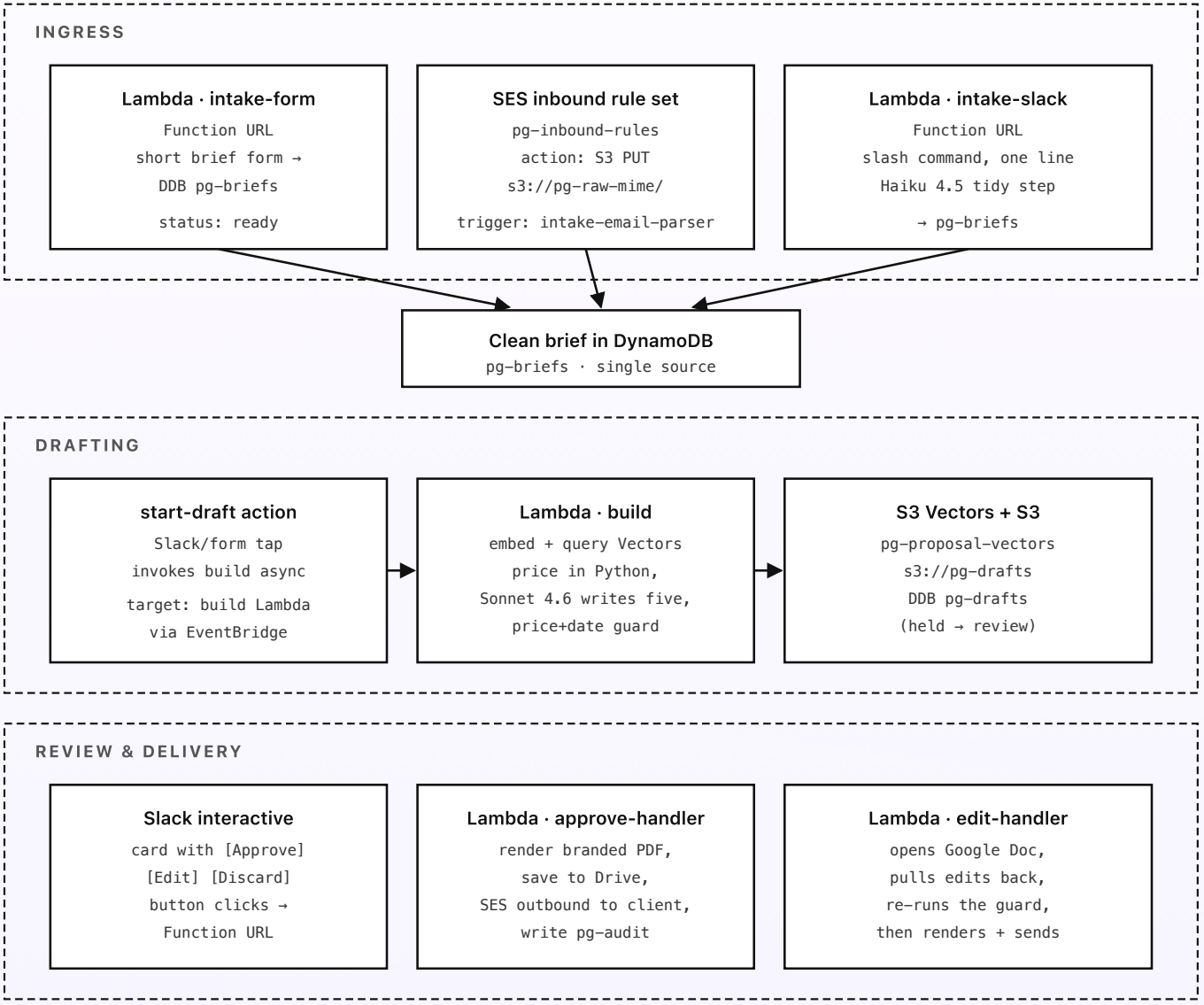
Engineering reference: the proposal generator architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the S3 Vectors retrieval setup, EventBridge 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 (Global profile), S3 Vectors, and EventBridge are all in good shape there. A second region for resilience isn't worth the setup at SMB volume — the failure mode for an SMB is a proposal that takes an extra hour, not a regional outage. One AWS account dedicated to the generator keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Nothing reaches a client without a human approving it — every action is logged to pg-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into one brief), drafting (the build Lambda retrieves, prices, drafts, and guards), review and delivery (the human approves and the proposal ships). Every Lambda is event- or action-driven; nothing is synchronous-chained.

Lambda functions

All Lambdas use the `arm64` architecture, the smallest memory size that meets latency targets, 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.

- **`intake-form`** — Lambda Function URL (`AuthType: NONE`, behind a shared-secret token in the form). Serves the short brief form and accepts its POST, writing a row to `pg-briefs` with `status: ready`. No model call. Memory: 256 MB. Timeout: 15 s.
- **`intake-email-parser`** — S3 PUT trigger on `s3://pg-raw-mime/`. Parses MIME, strips quoted replies and signatures, 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 a brief out of the thread as strict JSON, leaving unfound fields null. Writes to `pg-briefs` with `status: needs-review` and posts a Slack card. Memory: 512 MB. Timeout: 30 s.
- **`intake-slack`** — Lambda Function URL; verifies the Slack signing secret. Handles the `/proposal` slash command; runs the same Haiku tidy step on the one-line input and writes to `pg-briefs`. Memory: 256 MB. Timeout: 15 s.

- **`drive-sync`** — EventBridge Scheduler target, every 15 minutes. Uses the Google Drive + Docs APIs (service-account credentials in Secrets Manager under `pg/drive/sa`) to mirror the templates, past proposals, and rules docs to `s3://pg-templates-source/` when changed. On a new or changed past proposal, splits it into sections, embeds each with Titan Text Embeddings V2 (`amazon.titan-embed-text-v2:0`, 1024-dim), and upserts into the S3 Vectors index `pg-proposal-vectors`. Memory: 512 MB. Timeout: 60 s.
- **`build`** — invoked asynchronously via EventBridge when a brief's start-draft action fires. Reads the brief, embeds the need with Titan V2, queries `pg-proposal-vectors` for the nearest sections, computes the price summary in plain Python from the rate card in `s3://pg-templates-source/rules.txt`, assembles the grounded prompt, and makes one Claude Sonnet 4.6 call (`anthropic.claude-sonnet-4-6-20250115-v1:0` via `global.anthropic.claude-sonnet-4-6-20250115-v1:0`) returning five sections as structured JSON. Runs the price-and-date guard (pure Python), retries once on a structural failure, then writes the draft to `s3://pg-drafts/` and a row to `pg-drafts` with `status: ready-for-review` or `held`. Memory: 1024 MB. Timeout: 120 s.
- **`approve-handler`** — Lambda Function URL, public with `AuthType: NONE`; verifies a Slack signature on the request body. Triggered by Approve and Discard button clicks. On approve, renders the five sections to a branded PDF (a small HTML-to-PDF step), writes it to the Drive `Proposals/sent/` folder via the Drive API, sends it to the client via SES `SendRawEmail`, and writes a `sent` row to `pg-audit`. On discard, archives the brief and draft and writes a `discarded` row. Memory: 512 MB. Timeout: 30 s.

- **`edit-handler`** — Lambda Function URL. Triggered by the Edit button; ensures the linked Google Doc exists (creates it from the draft if needed) and returns its URL. On re-approve, pulls the doc text back via the Docs API, re-runs the price-and-date guard, and hands off to the same render-and-send path as `approve-handler`. A human-overridden price is logged as an override in `pg-audit` rather than blocked. Memory: 512 MB. Timeout: 30 s.
- **`digest`** — EventBridge Scheduler target, weekly Sunday 6pm. Reads `pg-drafts` and `pg-audit` for the past week; sends a Slack summary of briefs in, proposals sent, and drafts still open. No Bedrock; a plain summary table. Memory: 256 MB.
- **`summary`** — EventBridge Scheduler target, monthly on the first Monday at 9am. Reads the past month's `pg-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph pipeline narrative (briefs, sent, win-rate if outcomes are recorded); emails it via SES. Memory: 512 MB.

Storage

- **DynamoDB · `pg-briefs`** — one row per brief. PK `brief_id`; attributes: `client`, `need`, `budget`, `due_date`, `lane` (form/email/slack), `by_user`, `status` (ready/needs-review/drafting/done/archived). On-demand.
- **DynamoDB · `pg-drafts`** — one row per draft attempt. PK `(brief_id, draft_index)`; attributes: `status` (ready-for-review/held/sent/discarded), `price_total`, `doc_url`, `pdf_key`, `guard_result`. On-demand.
- **DynamoDB · `pg-audit`** — one row per write action of any kind. PK `(brief_id, ts)`; attributes: `action` (sent/edited/discarded/override), `by_user`, `before`, `after`. On-demand. No TTL — this is the long-term audit trail.

- **S3 Vectors** · `pg-proposal-vectors` — embeddings of each past-proposal section, 1024-dim, with metadata (proposal id, section type, client industry). Queried per draft for the nearest sections.
- **S3** · `pg-templates-source` — mirrored section templates, past proposals, and the rules/voice doc as plain text. Versioning enabled.
- **S3** · `pg-raw-mime` — raw inbound MIME from forwarded threads. Lifecycle to Glacier at 30 days; expiry at 7 years.
- **S3** · `pg-drafts` — the structured draft JSON and the rendered PDFs. Versioning enabled; lifecycle to Glacier at 90 days.

Bedrock

- **Draft model.** `anthropic.claude-sonnet-4-6-20250115-v1:0` via the Global cross-Region inference profile. One callsite: `build`, for writing the five sections. This is the only heavy call in the system, justified because writing a coherent multi-section document is real reasoning work.
- **Cheap model.** `anthropic.claude-haiku-4-5-20251001-v1:0` via the Global profile. Callsites: `intake-email-parser` and `intake-slack` (read a brief), Gate 1 voice touch-ups inside `build`, and `summary` (monthly narrative).
- **Embeddings.** `amazon.titan-embed-text-v2:0`, 1024-dim. Callsites: `drive-sync` (embed past sections on ingest) and `build` (embed the brief for retrieval).
- **Quotas.** Default account quotas are more than enough at SMB volume. The Sonnet draft fires once per started draft.

EventBridge config

- `pg-drive-sync` — `rate(15 minutes)`. Target: `drive-sync` Lambda.
- `pg-weekly-digest` — `cron(0 18 ? * SUN *)` in TZ. Target: `digest` Lambda.
- `pg-monthly-summary` — `cron(0 9 ? * 2#1 *)` (first Monday at 9am) in TZ. Target: `summary` Lambda.
- **Start-draft rule** — an EventBridge rule on the custom event `pg.start_draft` (emitted by the intake handlers when the owner taps start) with target `build`, so drafting runs async off the request that triggered it.

SES inbound and outbound

- Set the MX record on a dedicated subdomain (e.g. `proposals.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `pg-inbound-rules`: one rule with recipient `proposals@your-company.com` → spam scan → S3 PUT to `s3://pg-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-email-parser`.
- SES outbound for the approved proposals: verify a sender identity at `hello@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:

- **build role:** `dynamodb:GetItem` + `PutItem` on `pg-briefs` and `pg-drafts`; `s3:GetObject` on the templates and rules keys; `s3vectors:QueryVectors` on

`pg-proposal-vectors`; `bedrock:InvokeModel` on the Sonnet, Haiku, and Titan ARNs; `s3:PutObject` on `pg-drafts`.

- **approve-handler role:** `dynamodb:PutItem` on `pg-drafts` and `pg-audit`; `s3:GetObject` on `pg-drafts`; `ses:SendRawEmail` from the verified sender identity; `secretsmanager:GetSecretValue` on the Drive and Slack secrets; outbound network to `www.googleapis.com`.
- **edit-handler role:** `dynamodb:GetItem` + `PutItem` on `pg-drafts` and `pg-audit`; `secretsmanager:GetSecretValue` on the Docs-API service-account secret; outbound network to `docs.googleapis.com`; the same render-and-send permissions as `approve-handler`.
- **intake-email-parser role:** `s3:GetObject` on `pg-raw-mime`; `bedrock:InvokeModel` on the Haiku ARN; `dynamodb:PutItem` on `pg-briefs`; `secretsmanager:GetSecretValue` on the Slack webhook.
- **drive-sync role:** `secretsmanager:GetSecretValue` on the Google service-account secret; `s3:PutObject` on `pg-templates-source`; `s3vectors:PutVectors` on `pg-proposal-vectors`; `bedrock:InvokeModel` on the Titan ARN; outbound network to `www.googleapis.com`.

Slack interactive flow

Alert and draft messages are posted via the `chat.postMessage` Web API with Block Kit blocks containing the action buttons (Start draft on a brief card; Approve/Edit/Discard on a draft card). Button clicks are sent by Slack to the configured Interactivity request URL, which is the relevant handler Function URL. Each handler verifies the Slack signing secret on the inbound request, parses the

`action_id`, opens a modal where needed (Edit links the doc; the others are one-tap), and processes the response.

The Slack app needs `chat:write`, `commands` (for the slash command), and the Interactivity URL configured. The bot token lives in Secrets Manager under `pg/slack/bot-token`. The signing secret is `pg/slack/signing-secret`.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` + `"guard_held"` to a CloudWatch metric for alerting.
- **Alarms:** `build` failures > 0 in an hour; price-and-date guard holds > 10% of drafts in 24h (suggests the prompt or rate card drifted); approve-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:** $15/month threshold, alarm at 80% and 100%, posts to SNS topic `pg-cost-alarm` subscribed to the admin's email and Slack.

Config and secrets

Service-account credentials for the Drive, Docs, and Sheets APIs live in Secrets Manager under `pg/drive/sa` (one service account with scopes for all three). Slack bot token and signing secret under `pg/slack/*`. SES sender identity lives in IAM and the verified-domain config. The configured timezone, the rate-card location, the banned-claim list reference, and the brand assets (logo, terms page)

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

Deploy

GitHub Actions with OIDC into a short-lived deploy role — no long-lived AWS keys — running 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 `pg-templates-source` and `pg-drafts` so a bad Drive edit or a bad render can be rolled back, and keep the rate card under version control alongside the templates so a price change is reviewable. Total deployable surface: around nine Lambdas, three DDB tables, one S3 Vectors index, four S3 buckets, a handful of EventBridge 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.