7-PART SERIES · FREE COMPANION

Contract summarizer

A serverless reader that takes a long contract and hands back a one-page plain-English summary — who, what, money, key dates, renewal and cancellation terms — and flags the clauses worth a closer look, like auto-renewal, penalties, and liability. It quotes the exact clause behind every point, never gives legal advice, and says when to call a lawyer. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Contract summarizer

PART 1 OF 7

MAY 10, 2026 PART 1 OF 7 · CONTRACT SUMMARIZER SERIES ~5 MIN READ

A contract summarizer on AWS for a few dollars a month

A small business signs more contracts than anyone has time to read closely. The supplier agreement that auto-renews for two more years unless you cancel in a 30-day window nobody marked. The software order form with a late-fee clause buried on page nine. The lease, the service agreement, the NDA, the reseller deal with a liability cap that quietly puts your whole company on the hook. This post walks through the design of a small system that reads any of these, hands back a one-page plain-English summary, and flags the few clauses worth a closer look — without ever pretending to be your lawyer.

KEY TAKEAWAYS

- Three ways a contract comes in: a Drive folder, an inbox forwarding lane, and an e-sign webhook.
- Every contract runs the same flow: read the pages, pull the key terms, then flag the risky clauses.
- The summary has six blocks: parties, deal, money, key dates, renewal and cancellation, flagged clauses.
- Every point quotes the exact clause it came from. No quote, no point. It is never legal advice.
- 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.

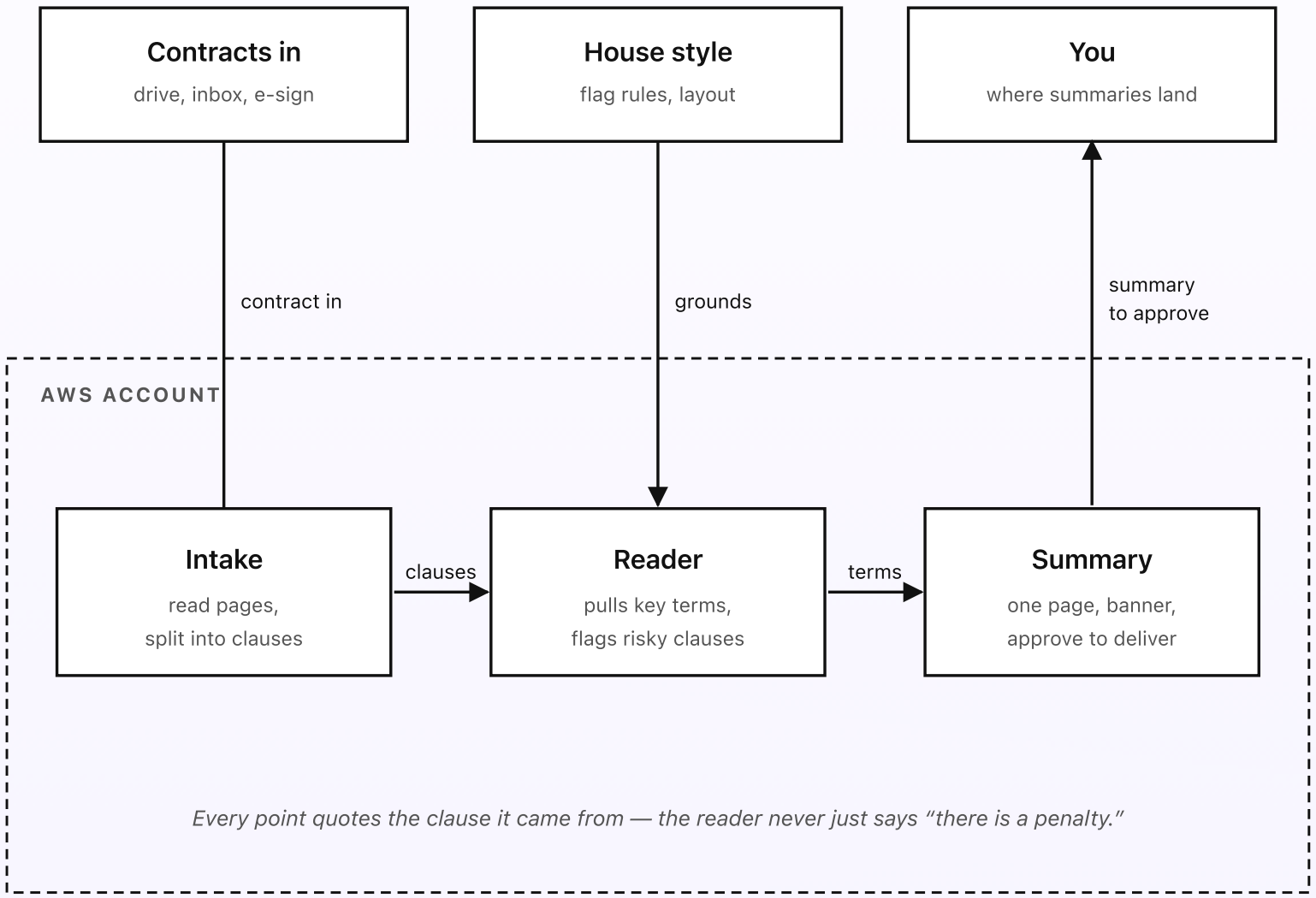

Contracts in
drive, inbox, e-sign
House style
flag rules, layout
You
where summaries land
contract in
grounds
summary
to approve
AWS ACCOUNT
Intake
read pages,
split into clauses
clauses
Reader
pulls key terms,
flags risky clauses
terms
Summary
one page, banner,
approve to deliver
Every point quotes the clause it came from — the reader never just says “there is a penalty.”

Fig 1. Three sources outside, three pieces inside AWS. Contracts flow in from a Drive folder, an inbox forwarding lane, and an e-sign webhook. The Reader pulls the key terms and flags the risky clauses. The Summary writes one plain-English page and waits for a person to approve it.

What you set up once (the outside)

- **Contracts in.** Three ways a contract reaches the system, covered in Part 2. A Drive folder you drop a file into. A dedicated inbox — forward a PDF to `review@your-company.com` and the system takes it from there. And a webhook from your e-sign tool (DocuSign, PandaDoc, and the like) that fires the moment an agreement is signed, so the summary is waiting for you before the ink is dry. You can use one lane or all three.
- **A house-style folder.** Two short Google Docs in a Drive folder. The *rules* doc says which clause types always get flagged — auto-renewal, penalties and late fees, liability and indemnity, personal guarantees, exclusivity, termination — and the thresholds that mean "tell the owner to call a lawyer" (for example, any liability cap above the contract value, or any personal guarantee at all). The *voice* doc holds the layout of the one-page summary and the exact wording of the not-legal-advice banner that sits at the top of every one.
- **You.** The person who reads the summary — usually the owner or the office manager. The summary lands wherever you asked: a Drive folder, an email, or a Slack channel. It arrives with an "Approve" button. Nothing is filed as final until a person taps it, so a summary is always a draft a human signed off on, never a machine's last word.

What runs on every contract (the inside)

- **The intake.** Takes the contract from whichever lane it arrived on and gets it ready to read. Amazon Textract turns the pages into text — it reads PDF, PNG, JPEG, and scanned documents, so a photographed contract works as well as a clean PDF. A small Python pass then splits the text into numbered clauses, so every later step can point at “clause 7.2” instead of a page somewhere. No model runs here; it is plain reading and splitting.
- **The reader.** Two passes. First, a Bedrock Haiku 4.5 call — Haiku is the cheap, fast model — reads the clauses and fills a fixed shape: parties, deal, money, key dates, renewal and cancellation. Each field carries the clause number it came from, or it is left blank. Second, a clause search finds the paragraphs that match the always-flag list, and a Bedrock Sonnet 4.6 call — the stronger model, used only where the stakes are higher — reads those few clauses and explains each in plain words: what it says, why it matters, and whether it is the kind of thing to run past a lawyer.
- **The summary.** Writes the six-block one-page summary, puts the not-legal-advice banner at the top, and lists the flagged clauses with the exact text quoted underneath each note. Then it stops and waits. The summary goes to your Drive, email, or Slack with an Approve button; tapping it files the summary against the contract and logs who approved it and when. Nothing is treated as final without that tap.

In plain words

A supplier emails you a renewal of your warehouse software agreement — eighteen pages, dense. You forward it to `review@your-company.com` and go back to your day. Four minutes later a summary lands in Slack. Parties: your company

and the vendor. Deal: the warehouse management software, same modules as last year. Money: $1,450/month, up 8%, net-30, 1.5% monthly late fee (clause 4.3). Key dates: starts July 1, runs 24 months, auto-renews for another 24 unless you give notice between 90 and 60 days before the end (clause 11.1). Flagged: *auto-renewal with a narrow notice window — clause 11.1 quoted — mark April 2 to May 2, 2028 now;* and *liability capped at one month's fees — clause 9.2 quoted — this is low for your exposure; worth a lawyer's eye before you sign.* You read it in two minutes, tap Approve, and forward the lawyer note to your attorney.

The cost of running this is about $3 a month at SMB volume. The cost of *not* running it is the auto-renewal nobody caught, the late-fee clause nobody read, or the liability cap that turns one bad shipment into a number with too many zeros.

DESIGN RULES THAT SHAPED EVERY DECISION

- Every point quotes the clause it came from. No quote, no point — the reader can't make things up.
- The same flow, always. Read the pages, pull the key terms, flag the risky clauses. There is no shortcut.
- It is never legal advice. Every summary carries a banner saying so, and high-stakes clauses say "call a lawyer."
- A person approves before anything is filed. The system drafts; a human signs off.
- The house-style folder lives in Drive. Changing a flag rule or the layout doesn't need a deploy.
- Every contract and every summary is logged and versioned. You can audit any reading later.

Why this shape

Most owners deal with a long contract in one of three ways: skim it and hope, pay a lawyer to read every page of a routine renewal, or sign it unread and find out later what was in there. Skimming misses the clause that mattered. Sending every routine document to a lawyer is slow and expensive for an NDA you've seen a hundred times. And signing unread is how a 30-day cancellation window slips past and a deal you wanted out of rolls over for another two years.

The setup above gives you a fast first read for everything. It pulls the handful of facts you actually need — who, what, money, dates — into one page, and it points a flashlight at the few clauses that tend to bite. It never tells you what to do, because that is a lawyer's job and the system knows it; it tells you what is there and when the stakes are high enough that a lawyer should look. The expensive human read is reserved for the contracts that earn it, and nothing gets signed because nobody had time to read it.

The next four posts walk through each piece in turn: how a contract gets read, how the key terms get pulled, how risky clauses get flagged, and how the whole summary stays grounded in the actual text. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 10, 2026 PART 2 OF 7 · CONTRACT SUMMARIZER SERIES ~4 MIN READ

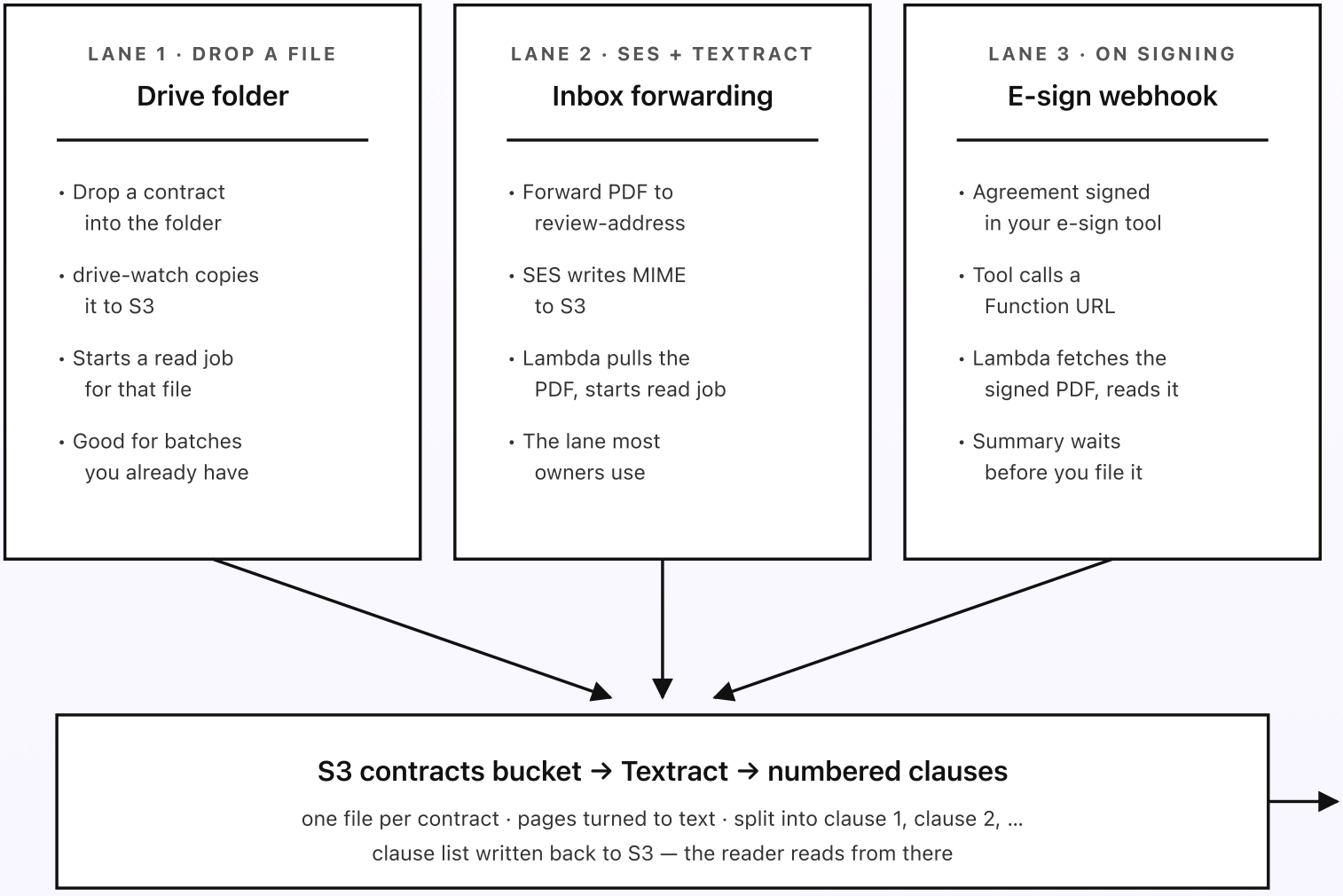
How a contract gets read

The summarizer only reads what reaches it. So the first job is making it easy to hand a contract over, in whatever form it already lives in. There are three ways one gets in: somebody drops a file in a Drive folder, somebody forwards a PDF to a dedicated address, or your e-sign tool pings the system the moment an agreement is signed. Once the file is in, Textract turns the pages into text and a small pass splits it into numbered clauses so everything that follows can point at exact lines.

KEY TAKEAWAYS

- Three intake lanes feed one reader: a Drive folder, an inbox-forwarding lane, and an e-sign webhook.
- Whatever lane it comes from, the contract lands as a file in S3 and kicks off the same read.
- Textract turns the pages into text — it reads PDFs, scans, and photos of a contract.
- A small Python pass splits the text into numbered clauses so later steps can cite exact lines.
- No model runs here. Reading and splitting is plain, cheap, and the same for every contract.

Three lanes into one reader



Every lane ends the same way — a file in S3 and a clause list — so the reader is one path.

Fig 2. Three lanes converge on one S3 bucket. From there the read is identical: Textract turns the pages into text and a Python pass splits it into numbered clauses. The reader never has to know whether the contract came from Drive, an email, or your e-sign tool.

Lane 1: the Drive folder

The simplest lane. You have a folder in Drive — call it `Contracts to review`. Drop a file in, and you're done. A small Lambda — `drive-watch` — checks the folder for new files, copies each one to `s3://cs-contracts/`, and starts a read job. This is the lane for the stack of agreements you already have on your laptop, or the renewal a supplier emailed you that you just save and drag in. It is also the easiest way to feed the system a batch on day one: select twenty old contracts, drop them in the folder, and come back to twenty summaries.

Because everything lands in the same S3 bucket no matter the lane, this folder is just a friendly front door. The work the system does on a file dropped here is exactly the work it does on a file that arrived any other way.

Lane 2: inbox forwarding (the lane most owners use)

Set up a dedicated inbound address — something like `review@your-company.com` — via Amazon SES. Anyone on the team forwards a contract PDF to that address and the system takes it from there. SES writes the raw email to `s3://cs-raw-mime/`. That S3 write triggers a Lambda that walks the email, finds the PDF attachment (or a Word or scanned document — Textract reads images and scans natively; a Word file falls back to a small text reader), copies it into the contracts bucket, and starts a read job.

This is the lane that fits how contracts actually arrive: as an attachment in an email, while you're busy doing something else. You forward and forget. A few minutes later the summary is waiting wherever you asked for it — covered in Part 5. No new tool to learn, no folder to remember; just a forward.

The system always reads the document a person handed it. It never goes looking for contracts on its own, and it never sends anything anywhere off the back of an intake — the only thing an intake produces is a draft summary that waits for you. That boundary matters: the system reads, it does not act.

Lane 3: the e-sign webhook

If your business signs through an e-sign tool — DocuSign, PandaDoc, and the like — the system can read the agreement the moment it's signed. You set up a webhook in the e-sign tool pointing at a Lambda Function URL (a plain web address that runs a small function; covered in the engineering reference). When an envelope completes, the tool calls that address. The Lambda fetches the signed PDF, copies it to the contracts bucket, and starts a read job — so the plain-English summary of what everyone just agreed to is sitting in your inbox before the celebration email.

This lane is the most hands-off of the three: nobody has to forward or drop anything, because the signing event itself is the trigger. It is also the one that catches the contract you might otherwise never re-read — the one you signed in a hurry and filed. A summary of it lands automatically, and if a clause should worry you, you find out now rather than at renewal.

Why one reader, not three

Three lanes in, but only one place where the reading actually happens. That's deliberate. If each lane did its own reading, a bug in how a forwarded PDF gets parsed would behave differently from the same bug on a dropped file, and every "why did this summary look wrong?" question would mean checking three code paths. Funneling everything down to one file in S3 means there is exactly one read step, one way clauses get numbered, and one place to look when something needs fixing. The lanes are just doors; the room behind them is the same.

Next post: how the reader takes those numbered clauses and pulls the key terms — parties, deal, money, dates — into a fixed shape, with every field tied back to the clause it came from.

PART 3 OF 7

MAY 10, 2026 PART 3 OF 7 · CONTRACT SUMMARIZER SERIES ~5 MIN READ

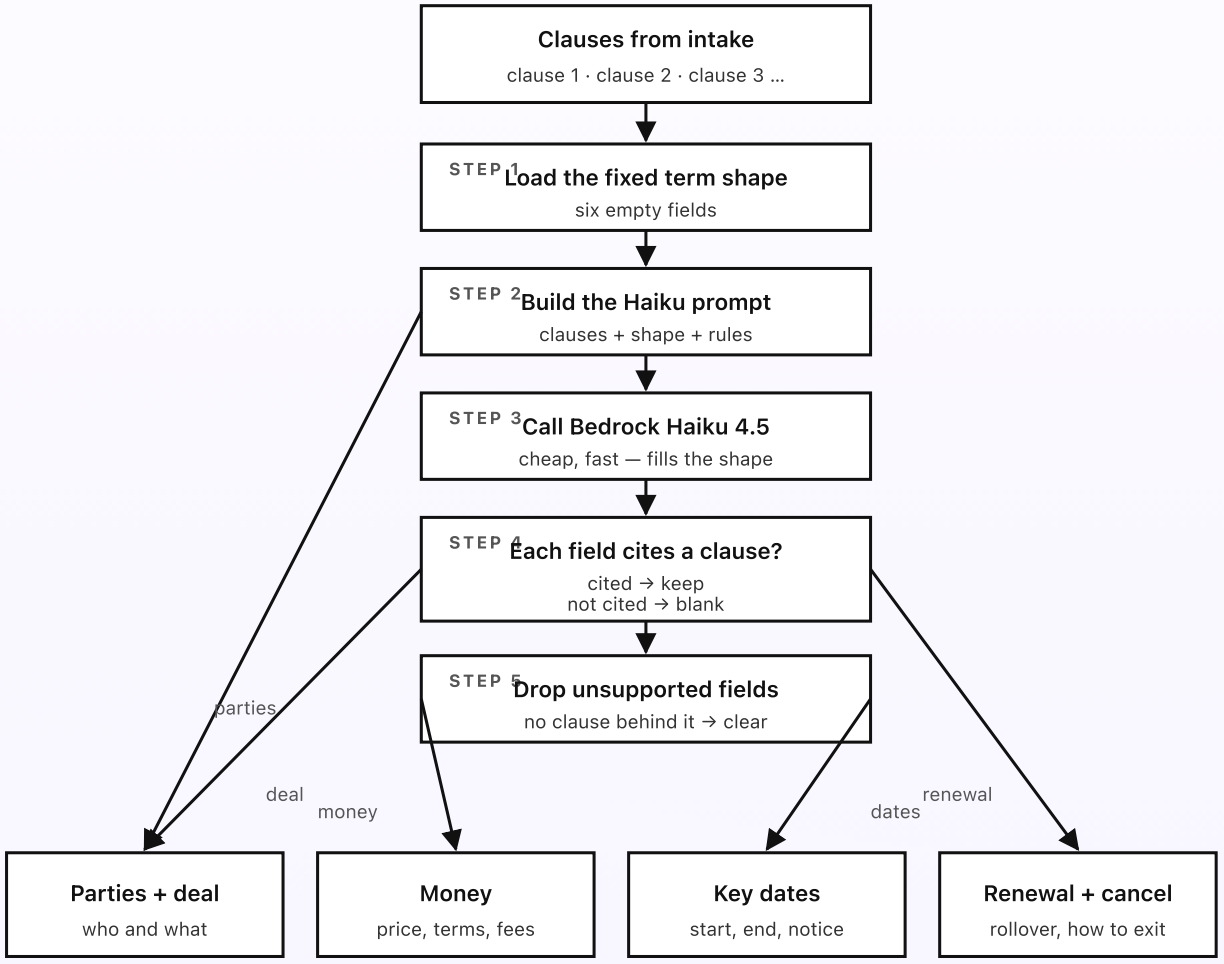
How the key terms get pulled

Now the system has the contract as a list of numbered clauses. The next job is to pull the handful of facts a busy owner actually needs — who is signing, what the deal is, the money, the dates, how it renews and how to get out — into a fixed shape. A single Bedrock Haiku 4.5 call does the reading. Haiku is the cheap, fast model; it is the right tool for fill-in-the-blanks work like this. The shape itself never changes; the model only fills it, and every field it fills has to name the clause it came from.

KEY TAKEAWAYS

- One Haiku 4.5 call fills a fixed shape of six fields. The shape is decided in code, not by the model.
- The six fields: parties, deal, money, key dates, renewal and cancellation, and a notes line.
- Every field carries the clause number it came from. A field with no clause is left blank, not guessed.
- The model is told plainly: do not invent a number or a date that isn't in the clauses.
- Haiku is the cheap path. The stronger Sonnet model is saved for the risk read in Part 4.

The fill flow, per contract



The shape is fixed in code — the model fills it, and a blank beats a guess.

Fig 3. The term pull, per contract. The shape of the summary is fixed in code; the Haiku call only fills the six fields, and every field has to name the clause it came from. A field with no clause behind it is left blank rather than guessed.

The fixed shape: six fields, decided in code

The summary always has the same six fields, in the same order, whether the contract is a two-page NDA or a forty-page master services agreement. *Parties*: who is signing with whom. *Deal*: what is being bought, sold, or agreed. *Money*: the price, the payment terms, any late fees. *Key dates*: when it starts, when it ends, and any notice windows. *Renewal and cancellation*: how it rolls over and how to get out. And a short *notes* line for anything that doesn't fit the first five but a reader would want.

The shape lives in the house-style doc, so you can reorder it or rename a field without touching code. What you can't do is let the model decide the shape. That's on purpose. If the model picked the fields, two contracts would come back looking different, and the value of a summary is that it always looks the same — you learn where to glance for the money and it's always there.

Cite the clause, or leave it blank

The single most important rule in this step: every field the model fills has to name the clause number it came from. The prompt is short and firm: "Fill each field from the clauses below. After each field, name the clause number you used. If a field isn't in the contract, leave it blank. Do not invent a number, a date, or a name that isn't in the text."

After the call, a plain Python check reads each field. If a field is filled but names a clause that doesn't exist, or names no clause at all, the field is cleared. A blank money field that reads "not stated in this contract" is honest and useful. A made-up payment term that looks right but isn't in the document is the kind of mistake that costs real money. So the system always prefers the blank. This is the same idea you'll see again in Part 5, applied here to the easy fields before the risky clauses get the stronger model in Part 4.

Why Haiku here, not the stronger model

Pulling the parties, the price, and the dates is fill-in-the-blanks work. The facts are usually stated plainly in the contract — "this Agreement is between X and Y," "the Fee is $1,450 per month" — and the job is to find them and copy them into the right field with the clause number attached. That is exactly what a cheap, fast model is good at. Haiku 4.5 does it in a second or two for a fraction of a cent, and the citation check catches the rare miss.

Saving the stronger Sonnet model for the risk read in Part 4 is a deliberate cost choice. The risky clauses — liability caps, auto-renewal traps, personal guarantees — need real reading-between-the-lines judgment, and there are only a few of them per contract. Spending the bigger model where the stakes are high and the cheap model where the facts are plain keeps the whole thing in coffee-money territory, which Part 6 lays out in full.

What comes out

The output of this step is a small term sheet written to S3: six fields, each with the text the model pulled and the clause numbers behind it. It is not the final summary yet — the risky clauses haven't been read and the not-legal-advice banner hasn't been added. But it is the backbone: the who, what, money, and dates that anyone reading the contract would want first, every one of them traceable to a line in the actual document.

Next post: how the system finds the few clauses worth a closer look, and how the stronger Sonnet model reads each one and says why it matters — and when to call a lawyer.

PART 4 OF 7

MAY 10, 2026 PART 4 OF 7 · CONTRACT SUMMARIZER SERIES ~5 MIN READ

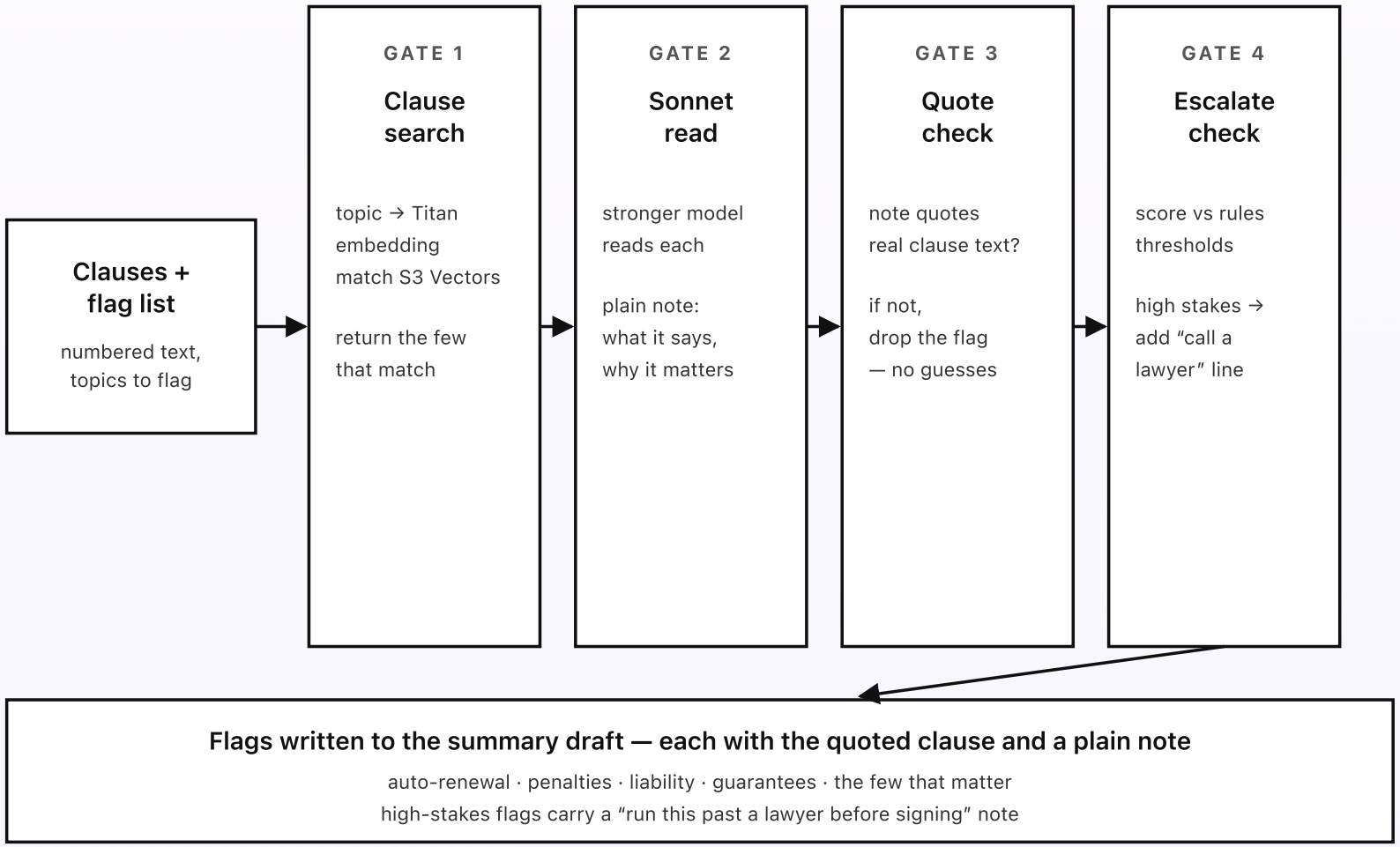
How risky clauses get flagged

The plain facts are pulled. Now the system has to find the few clauses that tend to bite — the auto-renewal with a tiny notice window, the late-fee that compounds, the liability cap that puts the whole company on the hook — and explain each in words an owner can act on. This is where the stronger Sonnet model earns its place. But it never reads the whole contract and it never tells you what to do. Four gates sit between a clause and a flag, and the last one decides whether the right next step is a lawyer.

KEY TAKEAWAYS

- A clause search finds the paragraphs that match the always-flag list, so the model reads only a few.
- Sonnet 4.6 — the stronger model — explains each flagged clause: what it says and why it matters.
- Every flag quotes the exact clause. A flag with no quote is dropped, same rule as the term pull.
- The model says what to look at, never what to do. It is not advice and the wording stays that way.
- A high-stakes clause trips the last gate: the flag says “run this past a lawyer before signing.”

Four gates on every flag



The model says what to look at and why — never what to do, and the high-stakes ones say call a lawyer.

Fig 4. Four gates between a clause and a flag. Search the clauses for the topics that matter. Read each with the stronger model. Check every note quotes real text. Score the stakes and, where they're high, say to call a lawyer. Only flags that pass all four reach the summary.

Gate 1: clause search

The system doesn't hand the whole contract to the stronger model — that would be slow and costly, and most clauses are boilerplate. Instead it searches. Each topic on the always-flag list (auto-renewal, penalties, liability, personal guarantees, exclusivity, termination) is turned into an embedding — a list of numbers that captures its meaning — using Amazon Titan Text Embeddings V2. Every clause from the intake is embedded the same way. A search in Amazon S3 Vectors, AWS's built-in vector store, matches each topic to the clauses that mean the same thing, even when the contract uses different words ("evergreen term" instead of "auto-renewal," "limitation of liability" instead of "liability cap").

The result is a short list: the handful of clauses that actually concern the topics you care about. Those are the only clauses the stronger model reads. A forty-page contract might surface six or eight clauses worth a look; the rest never reach Gate 2.

Gate 2: the Sonnet read

Now the stronger model goes to work, but only on those few clauses. Bedrock Sonnet 4.6 reads each matched clause and writes a short plain-English note: what the clause says, and why it matters to you, in two or three sentences. For an auto-

renewal clause: “This contract renews itself for another 24 months unless you cancel between 90 and 60 days before it ends. Miss that window and you’re locked in for two more years.” For a liability cap: “If something goes wrong, the most you could claim back from the vendor is one month’s fees — about $1,450 — no matter how big the actual loss.”

Sonnet is used here, and only here, because this is the reading that needs judgment. Spotting that a notice window is unusually short, or that a liability cap is low relative to what’s at stake, is exactly the kind of between-the-lines reading the cheap model can’t be trusted with. There are only a few clauses per contract, so the stronger model’s cost stays small — the math is in Part 6.

Gate 3: the quote check

Same rule as the term pull, applied to the flags: every note has to quote the exact text of the clause it’s about. After the Sonnet read, a plain Python check confirms the quoted text actually appears in that clause. If a note quotes nothing, or quotes words that aren’t in the clause, the whole flag is dropped — not shown with a warning, dropped. A flag you can’t trace back to the contract is worse than no flag, because it teaches the reader to doubt the ones that are real. The quote is what makes a flag checkable: the owner can read the clause for themselves, right there in the summary, and see the system isn’t making it up.

Gate 4: the escalate check

The last gate decides when a flag should send you to a lawyer. The rules doc holds the thresholds in plain prose: any personal guarantee at all, any liability cap

above the contract's value, any auto-renewal with a notice window shorter than 30 days, any clause that hands away exclusivity or your data. Each kept flag is scored against those thresholds. If it crosses one, the flag gets an extra line: "This is high-stakes — run it past a lawyer before you sign."

This is the system knowing its own limits. It will happily tell you what an indemnity clause says and why it matters. It will not tell you whether to accept it, negotiate it, or walk away — that is legal advice, and the system doesn't give legal advice. When the stakes are high enough that the answer really matters, it does the one responsible thing it can: it points you at a human who is allowed to answer. Part 5 makes that boundary — grounded, never advice, always a human in the loop — into the rule the whole summary lives by.

Next post: how the whole summary stays grounded in the actual text, the not-legal-advice banner, and the approval step before anything is filed or sent.

PART 5 OF 7

MAY 10, 2026 PART 5 OF 7 · CONTRACT SUMMARIZER SERIES ~5 MIN READ

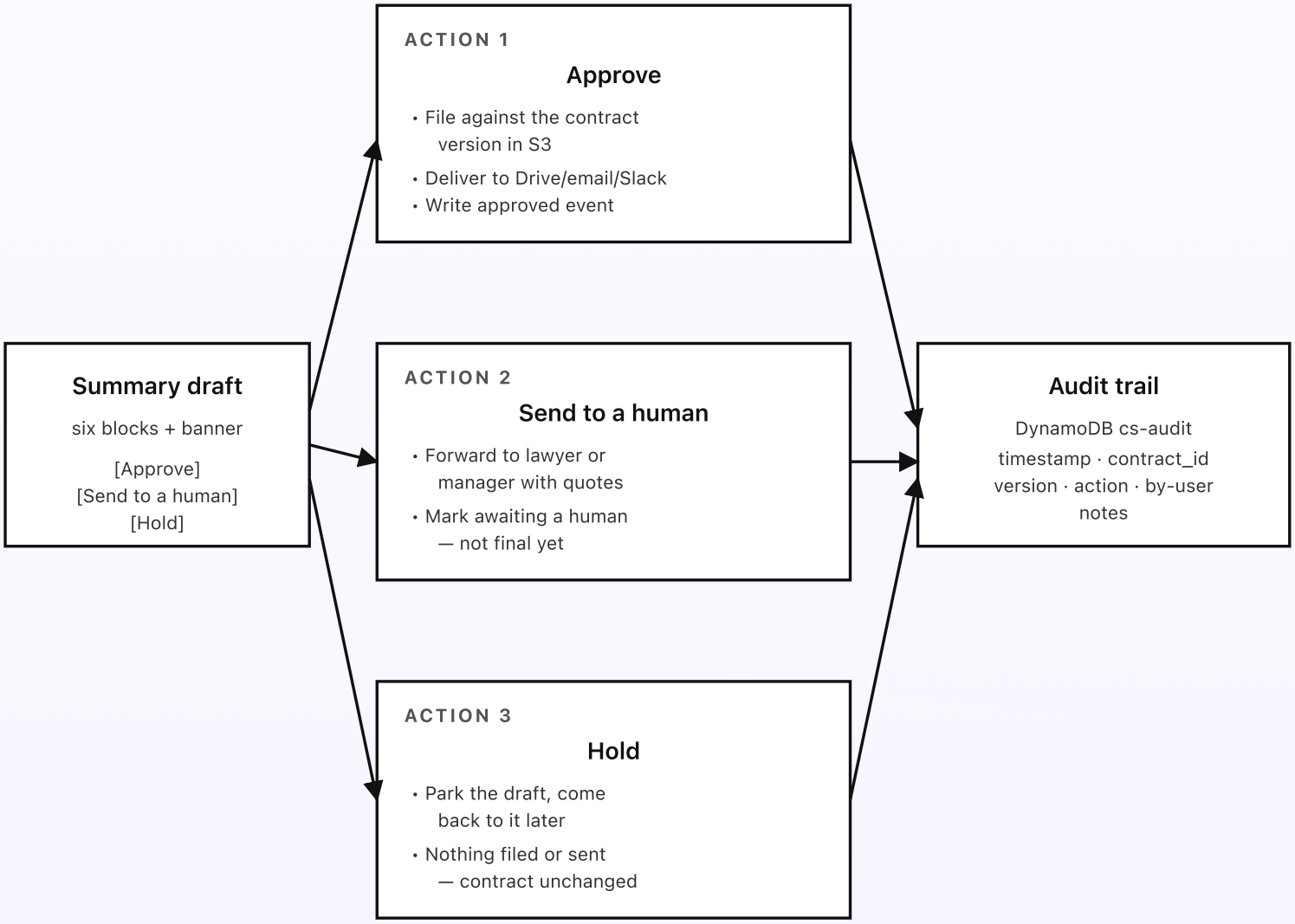
How a summary stays grounded

The terms are pulled and the risky clauses are flagged. Now the summary draft has to become something you can trust enough to act on — and that means three things have to hold. Every point traces back to the actual contract. Nothing in it pretends to be legal advice. And a person signs off before it's filed or sent anywhere. This post walks through the three things the system can do with a finished draft — approve, send to a human, or hold — and how the grounding, the banner, and the audit trail keep it honest.

KEY TAKEAWAYS

- Three things happen to a draft: *approve* (file and deliver), *send to a human* (a lawyer should see it), *hold* (park it).
- Grounding is the rule: every point quotes its clause, or it never reaches the draft.
- The not-legal-advice banner sits on top of every summary, no exceptions.
- A high-stakes clause routes the draft to a human before it counts as done.
- Every action is logged with the contract version, who acted, and when.

Three things on a finished draft



A summary is always a draft a human signed off on — the system reads and drafts, a person decides.

Fig 5. Three things an owner can do with a finished draft, three different effects. Approve files and delivers it. Send to a human routes it to a lawyer or manager. Hold parks it. Every action writes to the audit trail, and a high-stakes flag steers the draft toward a human before it counts as done.

Grounding: every point traces to the contract

You've seen the rule in Parts 3 and 4: every field in the term pull names its clause, and every risk flag quotes its clause. This is grounding, and it's the single thing that makes the summary worth trusting. A model left to its own devices will write a confident, fluent, plausible summary — and quietly slip in a payment term that isn't there, or a renewal date off by a month. Grounding makes that impossible to hide: if a point can't point at the text it came from, it never reaches the draft.

The practical effect is that the summary is checkable. Next to every number and every flag is the clause it came from, quoted. You don't have to take the system's word for the late-fee rate — clause 4.3 is right there. This is why the read step splits the contract into numbered clauses in the first place: so that "trust me" is never the answer. The answer is always "here's the line, read it yourself."

Never legal advice, on purpose

Every summary carries a banner across the top, in plain words: *"This is a plain-English reading of your contract, not legal advice. For anything that matters, talk to a lawyer."* It's not fine print and it's not optional — the summary writer adds it to every page, and there is no setting to turn it off.

The banner isn't just a disclaimer; it reflects how the system is actually built. The flags say what a clause *says* and why it *matters* — never what you should *do* about it. "This caps the vendor's liability at one month's fees" is a reading. "You should reject this clause" would be advice, and the system never crosses that line. The wording is checked: the summary writer is told to describe and explain, not to recommend, and the rare sentence that drifts into "you should" is rewritten or dropped. The point of the whole system is to make a busy owner a faster, better-informed reader — not to replace the lawyer they should call when it counts.

When the stakes are high, a human decides

The middle branch — *send to a human* — is the system's safety valve. It fires in two cases. First, automatically: when Gate 4 from Part 4 marked a flag high-stakes (a personal guarantee, an unusual liability cap, a short-notice auto-renewal), the draft is steered toward a human and isn't treated as final until one has seen it. Second, on demand: any owner can tap "Send to a human" on any summary, for any reason, and the draft plus its quoted clauses go to the configured lawyer or manager.

This is the human-in-the-loop rule made concrete. The system is allowed to read fast and draft well, but it is never allowed to be the last word on something that could put the business at real risk. On those, a person who is qualified to decide is always in the loop before anything counts as done. The system's job is to make sure that person sees the right contracts, with the risky clauses already quoted and explained, so their expensive time is spent deciding rather than hunting.

Every action logged, every version kept

The `cs-audit` table records every approve, send-to-human, and hold, with the contract id, the exact summary version, the user who acted, the timestamp, and any notes. S3 keeps every version of the contract and every version of the summary, so a summary approved in May can be pulled up next year exactly as it read then. If a contract is re-uploaded after an amendment, it gets a new version and a fresh summary; the old one stays on file.

This matters most for the contracts you'll only revisit once — at a dispute, at a renewal, at an audit. The next person to ask "what did we actually agree to?" gets the summary, the quoted clauses, and the record of who approved it. The audit trail is the memory the business keeps long after the person who signed has moved on.

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 stronger model doesn't break the bank.

PART 6 OF 7

MAY 10, 2026 PART 6 OF 7 · CONTRACT SUMMARIZER SERIES ~3 MIN READ

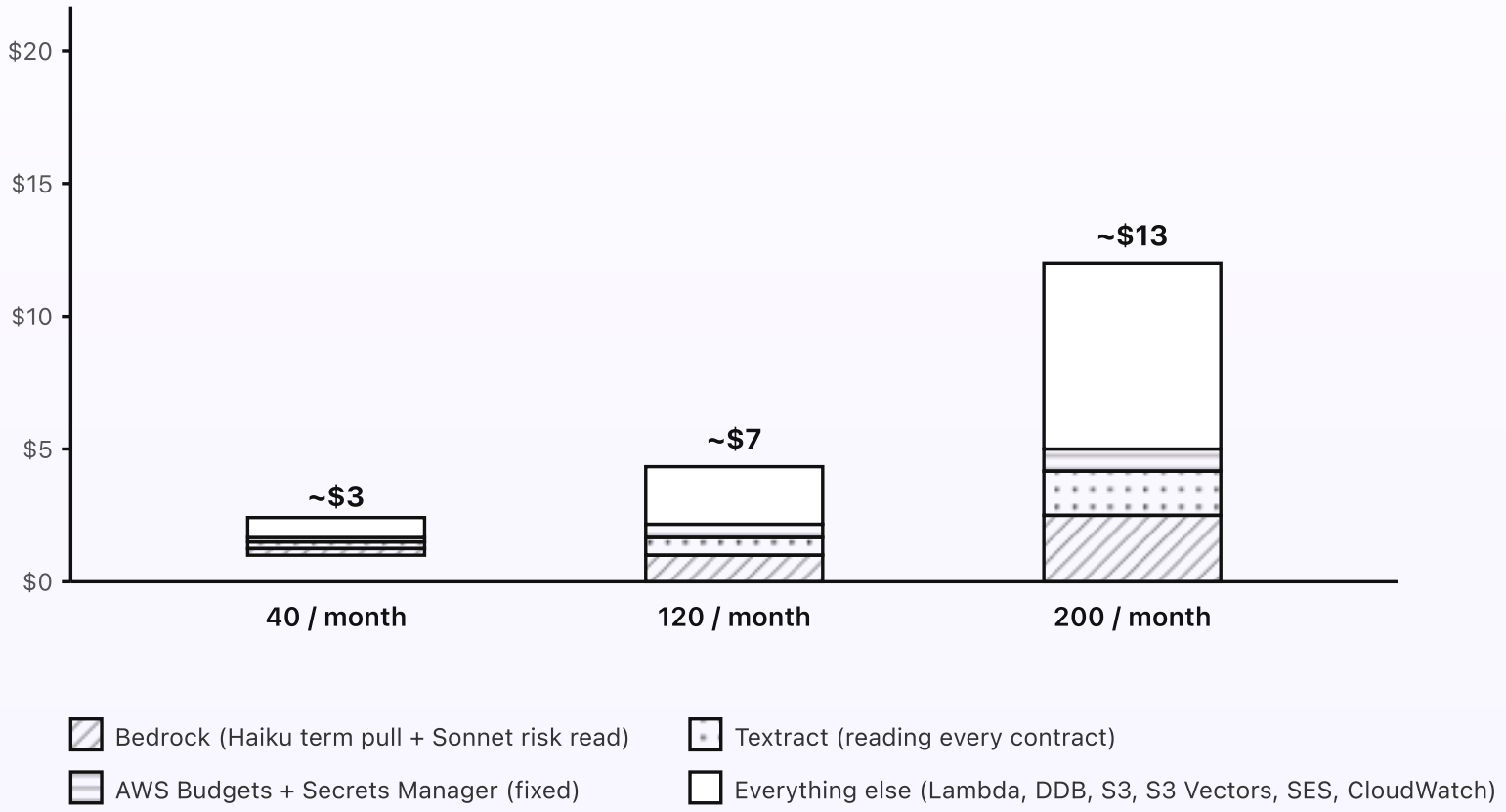
What the contract summarizer costs

The summarizer only spends money when a contract arrives. There's no daily tick, no always-on anything — the system sleeps until a file lands, reads it, and goes back to sleep. The cost per contract is the read step (Textract on the pages) plus two short model calls (Haiku to pull the terms, Sonnet to read the few risky clauses). At typical SMB volume that's a few dollars a month, fixed cost essentially zero.

KEY TAKEAWAYS

- Around $3/month at typical SMB volume (around 40 contracts read a month).
- Fixed AWS cost is essentially zero. No always-on compute, no NAT Gateway, no API Gateway.
- The cost is per contract — nothing runs when no contracts arrive.
- Each contract: Textract on the pages, one Haiku call, one Sonnet call on a few clauses.
- At 120 contracts a month the bill is around $7. At 200 it's around $13.

Cost at three volumes



The per-contract cost is the read plus two short model calls — a few cents per contract.

Fig 6. Monthly cost at three contract volumes. Bedrock and Textract grow with contract count because they fire once per contract; the everything-else bucket stays modest. Even at 200 contracts a month the bill lands around $13.

Where the dollars actually go

Bedrock (the bulk as volume grows). Two calls per contract. The Haiku 4.5 term pull reads the clauses and fills the six fields: a few thousand input tokens, a few hundred output, a fraction of a cent. The Sonnet 4.6 risk read sees only the few clauses that matched the flag search — not the whole contract — so even the stronger model handles a small slice of text per contract, a couple of cents. Together, a few cents per contract. That's the whole reason the system searches first and reads the strong model second: you pay the bigger model only for the paragraphs that earn it.

Textract. Per-page pricing; a typical contract is two to twenty pages. A few cents per contract to turn the pages into text. This grows linearly with how many contracts arrive, which is why the read slice tracks contract count on the chart.

Lambda runtime. The intake, the reader, the summary writer, and the approve/Function URL handlers each run for a second or two per contract. Pennies a month at all three volumes — nothing runs except when a contract is being processed.

DynamoDB on-demand. Two small tables: `cs-jobs` (one row per contract read) and `cs-audit` (one row per approve, send-to-human, or hold). A handful of reads and writes per contract. Pennies.

S3 + S3 Vectors. The raw contracts, the cleaned text, and the summaries, all versioned; plus the clause embeddings in the S3 Vectors index used by the flag search. A few MB per contract at most. Effectively free at SMB volume.

SES. Inbound for the forwarding lane and outbound for delivered summaries: $0.10 per thousand messages each way. A couple of cents a year at this scale.

What doesn't cost money

- **API Gateway.** Replaced by Lambda Function URLs for the e-sign webhook 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 system runs only when a contract arrives.
- **A daily tick.** Unlike a watcher, there's nothing to poll. Cost is purely per contract.
- **The strong model on everything.** Sonnet reads only the few flagged clauses, never the whole contract. The cheap Haiku model does the plain term pull.

How the cost scales

The bill tracks one number: how many contracts you read. Bedrock and Textract are per-contract, so they grow roughly linearly. The everything-else bucket grows slowly because the work per contract is small and the storage is cheap. So 400 contracts a month lands around $25, and 800 around $48. Longer contracts cost a little more (more pages to read, more clauses to search), but the strong-model cost stays bounded because it only ever sees the flagged clauses, not the full document.

Set an AWS Budgets alarm at $15/month so anything unusual — a stuck retry loop, an accidental batch of a thousand files — pages you before the bill matters. The summarizer'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, the S3 Vectors index, Bedrock model IDs, and the DynamoDB schemas.

PART 7 OF 7

MAY 10, 2026 PART 7 OF 7 · CONTRACT SUMMARIZER SERIES ~8 MIN READ

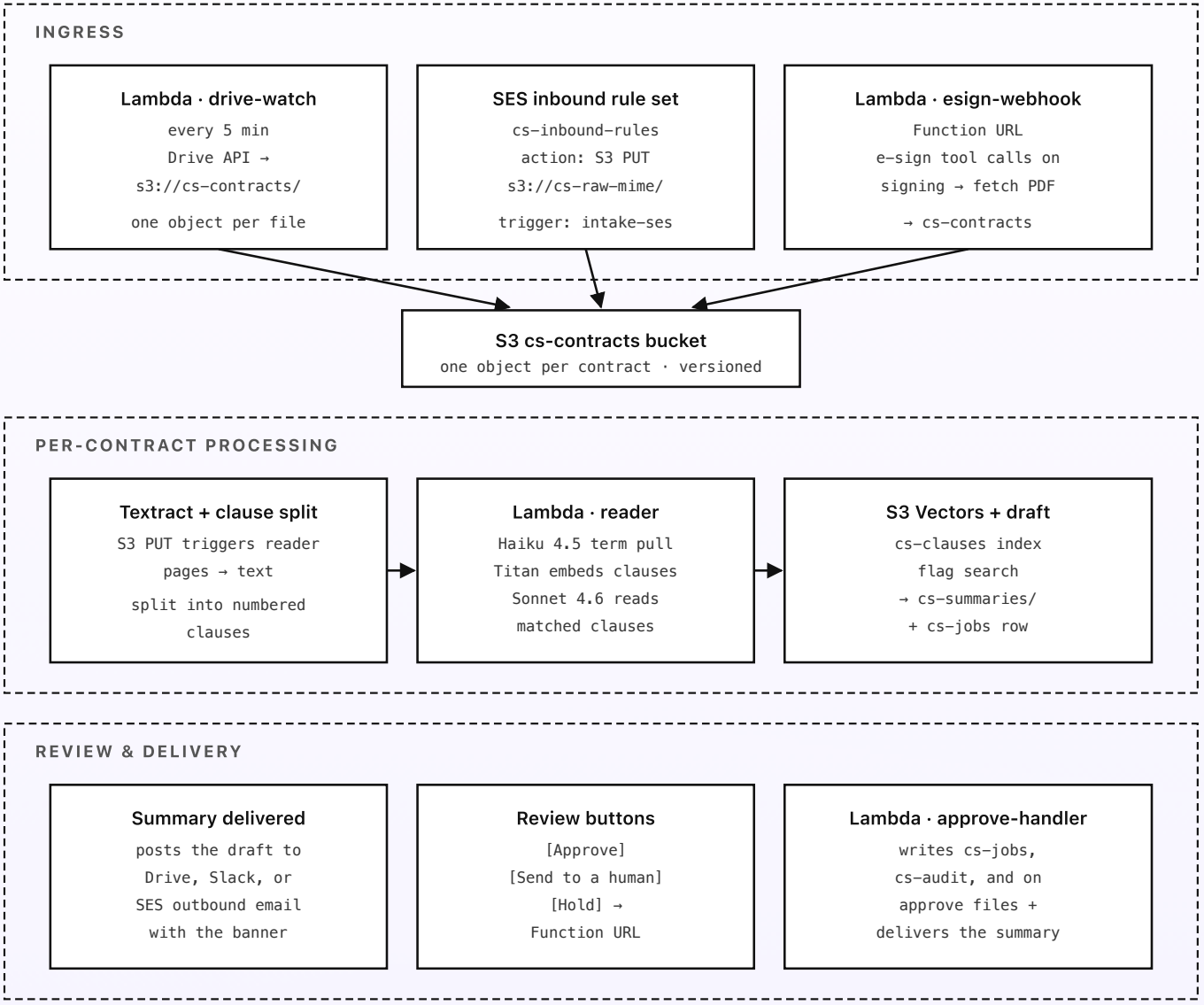
Engineering reference: the contract summarizer architecture

Same system, drawn for engineers. Region, service names, resource identifiers, Bedrock model IDs, Lambda inventory, IAM scopes, the SES inbound rule set, the S3 Vectors index, the DynamoDB schemas, and the approval 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, Textract, Bedrock cross-Region inference, and S3 Vectors are all available there. A second region for resilience isn't worth the setup at SMB volume — the failure mode is a contract that has to be re-forwarded, not a regional outage. One AWS account dedicated to the summarizer (separate from other workloads) keeps the IAM blast radius small and lets a single AWS Budgets alarm cover the whole system.

Topology



Every point quotes its clause — and every action is logged to cs-audit.

Fig 7. AWS topology, in three regions of the diagram: ingress (three lanes into the contracts bucket), per-contract processing (read, term pull, flag search, risk read, draft), review and delivery (the summary ships and the owner's decision is recorded). Every Lambda is event-driven; nothing is synchronous-chained beyond a single contract's read.

Lambda functions

All Lambdas use the `arm64` architecture (Graviton), 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.

- **`drive-watch`** — EventBridge Scheduler target, fires every 5 minutes. Uses the Google Drive API (service-account credentials in Secrets Manager under `cs/drive/sa`) to list new files in the watched folder and copy each to `s3://cs-contracts/<uuid>`. Records the Drive file id in `cs-jobs` to avoid re-importing. Memory: 256 MB. Timeout: 30 s.
- **`intake-ses`** — S3 PUT trigger on `s3://cs-raw-mime/`. Walks the MIME tree, extracts the contract attachment (PDF/PNG/JPEG/TIFF go straight to Textract; DOCX falls back to `python-docx`, XLSX to `openpyxl`), and copies it to `s3://cs-contracts/`. Both fallback packages are stable and widely used in 2026, though maintenance velocity is light — acceptable for a path that runs a few dozen times a month; `python-docx-oss` is a drop-in if extraction precision becomes a concern. Memory: 512 MB. Timeout: 60 s.
- **`esign-webhook`** — Lambda Function URL, `AuthType: NONE`; verifies an HMAC signature from the e-sign provider (DocuSign Connect, PandaDoc webhooks,

etc.) using the secret in `cs/esign/hmac`. On a completed-envelope event, fetches the signed PDF via the provider API and copies it to `s3://cs-contracts/`. Memory: 256 MB. Timeout: 30 s.

- **`reader`** — S3 PUT trigger on `s3://cs-contracts/`. The core function. Runs Textract `StartDocumentTextDetection` + `StartDocumentAnalysis` asynchronously for multi-page contracts; on completion (SNS notification) reads the structured text and splits it into numbered clauses with a deterministic Python pass. Calls Bedrock Haiku 4.5 (`anthropic.claude-haiku-4-5-20251001-v1:0` via `global.anthropic.claude-haiku-4-5-20251001-v1:0`) to fill the fixed term shape, validating that each field cites an existing clause. Embeds each clause with Titan Text Embeddings V2 (`amazon.titan-embed-text-v2:0`, 1024-dim) and upserts to the `cs-clauses` S3 Vectors index; runs the always-flag topic search; calls Bedrock Sonnet 4.6 (`anthropic.claude-sonnet-4-6-20250930-v1:0` via `global.anthropic.claude-sonnet-4-6-20250930-v1:0`) on matched clauses for the risk flags; validates every flag quotes real clause text; writes the summary draft to `s3://cs-summaries/` and a row to `cs-jobs`. Memory: 1024 MB. Timeout: 120 s.
- **`deliver`** — EventBridge rule on the `cs.summarize_done` event. Posts the draft to the configured surface: a Drive folder (Drive API), a Slack channel (`chat.postMessage` with Block Kit buttons), or an email (SES `SendRawEmail`), each with the not-legal-advice banner and the Approve / Send-to-a-human / Hold actions. Memory: 256 MB. Timeout: 30 s.
- **`approve-handler`** — Lambda Function URL, `AuthType: NONE`; verifies a Slack signature (or a signed token for email-link clicks). Triggered by the three review buttons. Writes to `cs-jobs` and `cs-audit`; on *approve*, files the summary

against the contract version and delivers it; on *send-to-a-human*, forwards the draft and quoted clauses to the configured lawyer/manager via SES and marks the job `awaiting_human`; on *hold*, parks the draft. Memory: 256 MB. Timeout: 15 s.

- **`summary-monthly`** — EventBridge Scheduler target, first Monday 9am. Reads the past month's `cs-jobs` and `cs-audit`; calls Bedrock Haiku 4.5 to write a one-paragraph rollup (how many contracts read, how many high-stakes flags, how many sent to a human); emails it via SES to the configured stakeholders. Memory: 512 MB.

Storage

- **DynamoDB · `cs-jobs`** — one row per contract read. PK `contract_id`; sort key `summary_version`; attributes: `source_lane` (drive/ses/esign), `status` (reading/drafted/approved/awaiting_human/held), `page_count`, `flag_count`, `high_stakes` (bool), `created_at`. On-demand.
- **DynamoDB · `cs-audit`** — one row per review action. PK `(contract_id, ts)`; attributes: `action` (approve/send_to_human/hold), `by_user`, `summary_version`, `notes`. On-demand. No TTL — long-term audit trail.
- **S3 · `cs-contracts`** — raw contracts, one object per upload. Versioning enabled. Lifecycle to Glacier at 90 days; expiry at 7 years.
- **S3 · `cs-summaries`** — the cleaned clause text and the one-page summary drafts/finals, versioned, so any summary can be pulled up exactly as it read when approved.

- **S3 · `cs-raw-mime`** — raw inbound MIME from the forwarding lane. Lifecycle to Glacier at 30 days; expiry at 7 years.
- **S3 Vectors · `cs-clauses`** — clause embeddings (1024-dim Titan V2), keyed by `(contract_id, clause_no)` with the clause text as metadata. Used by the always-flag topic search; entries for a superseded contract version are pruned on re-upload.

Bedrock

- **Foundation models.** `anthropic.claude-haiku-4-5-20251001-v1:0` for the term pull and the monthly rollup, and `anthropic.claude-sonnet-4-6-20250930-v1:0` for the risk read — both via their Global cross-Region inference profiles (`global.anthropic.*`). Haiku is the cheap path; Sonnet is reserved for the few matched clauses where reasoning earns its cost.
- **Embeddings.** `amazon.titan-embed-text-v2:0` (1024-dim) for clause embeddings, stored in the `cs-clauses` S3 Vectors index. This is the one system in the philosophy that genuinely needs retrieval — the flag search has to find topically-matching clauses regardless of the contract's wording.
- **Quotas.** Default account quotas are more than enough at SMB volume. Both models fire at most twice per contract; at a couple hundred contracts a month that's well within limits.

Grounding and guardrails

- **Citation enforcement.** The `reader` validates, in plain Python, that every term field cites an existing clause number and every risk flag quotes text that literally

appears in its clause. Unsupported fields are blanked; unsupported flags are dropped. No model output reaches the draft un-checked.

- **Not-legal-advice banner.** Prepended to every summary by the summary writer; no config flag disables it. The Sonnet prompt is instructed to describe and explain, never to recommend; a lint pass rewrites or drops any sentence that drifts into prescriptive "you should" phrasing.
- **Human-in-the-loop.** A summary is never `approved` by the system. High-stakes flags (rules-doc thresholds in `/cs/config/flag-rules`) set `high_stakes=true` and route the draft to `awaiting_human` until a person acts. The owner can also send any draft to a human on demand.

EventBridge Scheduler and SES

- **`cs-drive-watch`** — `rate(5 minutes)`. Target: `drive-watch` Lambda.
- **`cs-monthly-summary`** — `cron(0 9 ? * 2#1 *)` (first Monday 9am) in `TZ_NAME`. Target: `summary-monthly` Lambda.
- Set the MX record on a dedicated subdomain (e.g. `review.your-company.com`) to `inbound-smtp.ap-southeast-1.amazonaws.com`.
- SES inbound rule set `cs-inbound-rules`: one rule with recipient `review@your-company.com` → spam scan → S3 PUT to `s3://cs-raw-mime/<message-id>` → stop. The S3 PUT triggers `intake-ses`.
- SES outbound for delivered summaries and lawyer hand-offs: verify a sender identity at `summaries@your-company.com` with DKIM and SPF. Out of sandbox by request.

IAM (least privilege per Lambda)

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

- **reader role:** `s3:GetObject` on `cs-contracts`; `s3:PutObject` on `cs-summaries`; `textract:StartDocumentTextDetection` + `StartDocumentAnalysis` + `GetDocument*`; `bedrock:InvokeModel` on the Haiku, Sonnet, and Titan ARNs; `s3vectors:PutVectors` + `QueryVectors` on `cs-clauses`; `dynamodb:PutItem` on `cs-jobs`; `events:PutEvents` for the done event.
- **deliver role:** `s3:GetObject` on `cs-summaries`; `secretsmanager:GetSecretValue` on the Slack and Drive secrets; `ses:SendRawEmail`; outbound network to `slack.com` and `www.googleapis.com`.
- **approve-handler role:** `dynamodb:PutItem` on `cs-jobs` and `cs-audit`; `s3:GetObject` / `PutObject` on `cs-summaries`; `ses:SendRawEmail` for the human hand-off; `secretsmanager:GetSecretValue` on the Slack signing secret.
- **intake-ses role:** `s3:GetObject` on `cs-raw-mime`; `s3:PutObject` on `cs-contracts`.
- **drive-watch and esign-webhook roles:** `secretsmanager:GetSecretValue` on the relevant secret; `s3:PutObject` on `cs-contracts`; `dynamodb:PutItem` on `cs-jobs`; outbound network to the Google or e-sign API host.

Review and approval flow

Summaries delivered to Slack use `chat.postMessage` with Block Kit blocks carrying the three action buttons; clicks hit the configured Interactivity request URL, which is the `approve-handler` Function URL. The handler verifies the Slack signing secret, parses the `action_id` (`approve`, `send_to_human`, `ack_hold`), and processes it. Email-delivered summaries carry signed links to the same handler. The Slack app needs `chat:write` and the Interactivity URL configured; the bot token lives in Secrets Manager under `cs/slack/bot-token`, the signing secret under `cs/slack/signing-secret`.

Observability and cost gates

- **CloudWatch Logs:** all Lambdas, 7-day retention, structured JSON. Subscription filter on `"error"` + `"throttle"` + `"timeout"` to a metric for alerting.
- **Alarms:** `reader` failures > 0 in an hour (the read is the one piece that has to work); Textract job failures > 1%; approve-handler signature-verification failures > 5/hour (a rotated Slack secret).
- **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 `cs-cost-alarm` subscribed to the on-call admin's email and Slack.

Config and secrets

Drive service-account credentials live in Secrets Manager under `cs/drive/sa`; the e-sign HMAC secret under `cs/esign/hmac`; Slack bot token and signing secret under `cs/slack/*`. The always-flag list, the escalate-to-a-lawyer

thresholds, the summary layout, the banner wording, the delivery surface, and the lawyer/manager hand-off address all live in Parameter Store under `/cs/config/` (mirrored from the house-style Drive docs by a small sync, so a non-engineer can edit a flag rule without a deploy). Lambdas fetch config on cold start and cache for the execution environment's lifetime.

Deploy

GitHub Actions with OIDC into a deploy role (no long-lived keys), building and shipping with 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 `cs-contracts` and `cs-summaries` so any version of a contract or summary can be pulled up later, and keep the two Function URLs (`esign-webhook`, `approve-handler`) signature-verified rather than public-open. Total deployable surface: around seven Lambdas, two DynamoDB tables, three S3 buckets, one S3 Vectors index, 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.