7-PART SERIES · FREE COMPANION

Website chat assistant

A serverless chat widget that lives on your website, answers visitors from your own knowledge in real time, hands the rest to a human cleanly, and turns every miss into a better answer next week. Seven posts on the same system — one diagram at a time — with an engineering reference at the end.

CONTENTS

Website chat assistant

PART 1 OF 7

MAY 1, 2026 PART 1 OF 7 · WEBSITE CHAT ASSISTANT SERIES ~5 MIN READ

A website chat assistant on AWS for a few dollars a month

A visitor lands on your site at 9pm with a question. They don’t want to fill out a form, they don’t want to email and wait, and they definitely don’t want to read your FAQ. Here’s how to design a small chat widget that answers them from your own knowledge in real time, hands the rest to you cleanly, and quietly logs every question it couldn’t answer so the assistant gets smarter every week.

KEY TAKEAWAYS

- Three external surfaces, three AWS pieces. Visitor opens the widget, the cloud answers from your Drive knowledge, hands off when it shouldn't guess.
- Every visitor turn ends in one of four moves: answer, clarify, hand off, or decline. No fifth.
- No citation, no auto-answer. Above the confidence line the assistant must point to a passage from your docs; below it routes to clarify or hand off.
- Short-term memory only. The session knows the last few turns; it does not remember the visitor next week unless they sign in.
- Runs on AWS for about $3/month at typical SMB volume; cents per conversation, dominated by Bedrock tokens for the answerer.

The whole system on one page

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

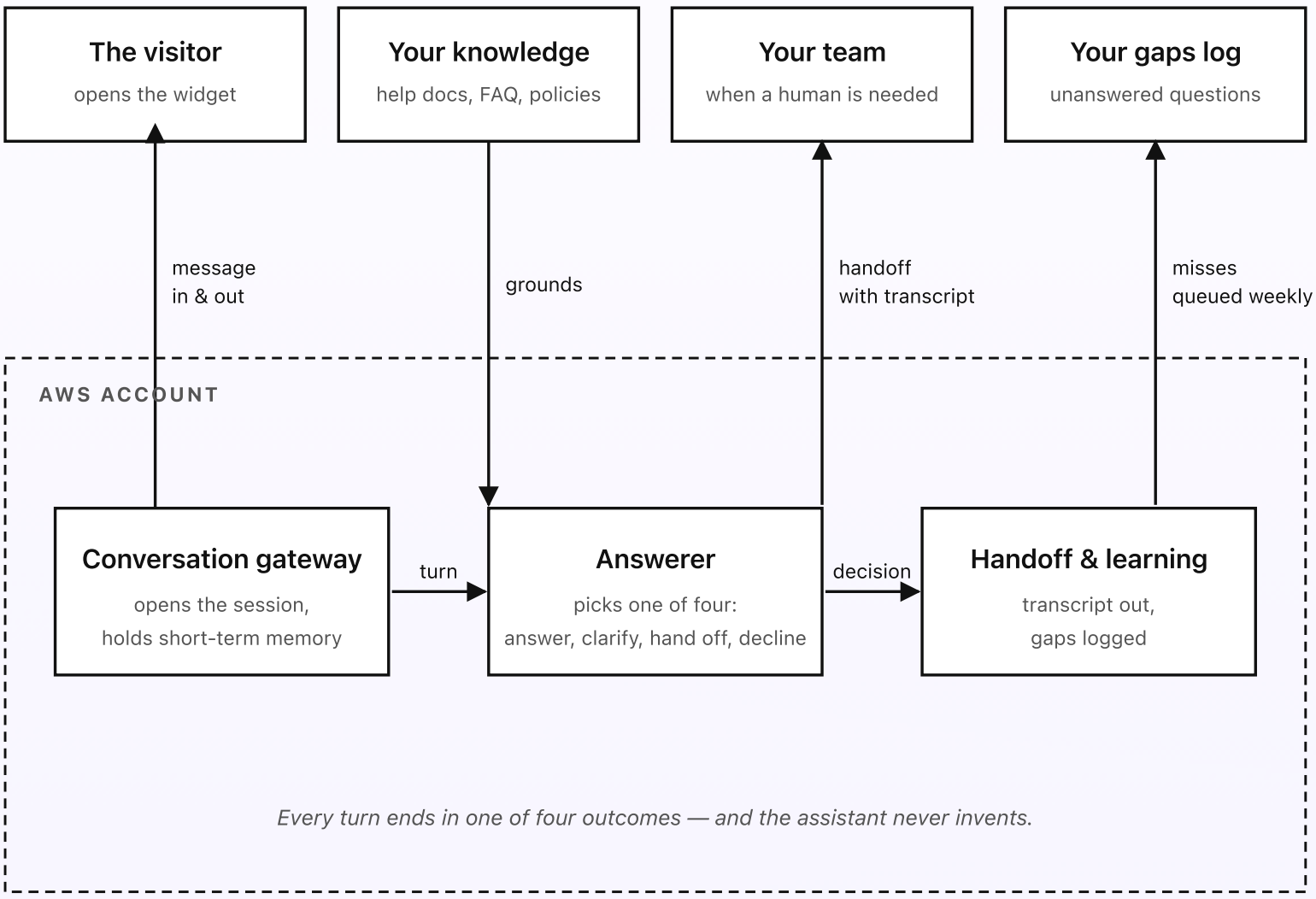

The visitor
opens the widget
Your knowledge
help docs, FAQ, policies
Your team
when a human is needed
Your gaps log
unanswered questions
message
in & out
grounds
handoff
with transcript
misses
queued weekly
AWS ACCOUNT
Conversation gateway
opens the session,
holds short-term memory
turn
Answerer
picks one of four:
answer, clarify, hand off, decline
decision
Handoff & learning
transcript out,
gaps logged
Every turn ends in one of four outcomes — and the assistant never invents.

Fig 1. Four outside surfaces, three pieces inside AWS. Visitor opens the widget, the Answerer replies from your knowledge or hands off, and every miss feeds back into a weekly review.

What you set up once (the outside)

- **A small embed snippet** — a few lines of script you paste into your site template once. The widget bubble appears in the corner; clicking it opens a chat panel that reuses your site's own font and colour. The snippet does nothing on its own — the work happens after the websocket opens.
- **A knowledge folder** — the help docs, FAQ, policies, pricing, hours, return rules, and anything else you'd want a new hire to read on day one. Lives in a Google Drive folder; you edit a doc, the assistant picks up the change on its next refresh, no deploy.
- **A handoff destination** — the inbox, Slack channel, or shared queue you already use. The assistant packages the transcript, adds a one-line summary of what the visitor wanted, and drops it where you'll see it.
- **A gaps log** — a small Drive doc or sheet that fills up with the visitor turns the assistant couldn't confidently answer. It's the to-do list for next week's knowledge updates.

What runs on every conversation (the inside)

- **The conversation gateway** — accepts the websocket from the widget, opens a session keyed to a short-lived token, and holds the last few turns of context so follow-up questions don't need to repeat the whole story. Idle conversations time out cleanly.

- **The answerer** — on every visitor turn, searches your knowledge for relevant passages, hands the cleanest passages to a small AI, and asks for one of four moves: answer with a citation, ask one short clarifying question, hand off to a human, or politely decline (off-topic, unsafe, or out of remit). The answer streams back word by word so the visitor isn't staring at a spinner.
- **Handoff and learning** — when the answerer picks "hand off," the transcript and a short summary go to your inbox or Slack. When confidence is low or no useful passage was found, the turn lands in the gaps log instead, with a timestamp and the question. You batch-review weekly.

In plain words

A visitor opens the chat. The widget connects to the cloud, the cloud loads a tiny scratchpad for the conversation, and the visitor types "do you ship to Canada?" The cloud searches your help docs, finds the shipping policy, and streams back a one-line answer with a small "from: shipping policy" citation. The visitor follows up — "how long does it take?" — and the cloud uses the same scratchpad, no need to repeat "to Canada." Two turns later they ask something the docs don't cover; the assistant doesn't guess. It says it'll get a human and drops the transcript in your inbox. The unanswered turn lands in your gaps log so you can write a paragraph about it and the next visitor won't have to wait.

Total cost runs in coffee-money territory at typical small-business volume — cents per conversation, going up smoothly with how often the widget gets opened.

DESIGN RULES THAT SHAPED EVERY DECISION

- The assistant answers from your knowledge file only — never invents. No citation, no auto-answer.
- Streaming first. The visitor sees words within a second; spinners on a chat widget are how visitors give up.
- Short-term memory only. The session knows the last few turns; it does not remember the visitor next week. No long-lived profiles unless the visitor signs in.
- Confidence gates the route. High-confidence with citation auto-answers; borderline becomes a clarifying question or a handoff; off-topic gets a polite decline.
- Configuration lives in a Drive folder. Updating the FAQ, hours, or return policy never needs a deploy.
- Misses are not failures — they are the input to next week's knowledge update.

Why this shape

Most chat widgets fall into one of two traps. The first kind is a generic AI bot — happy to answer anything, including things about your business that aren't true. The second kind is a fancy contact form — every message becomes a ticket in your queue, even the ones with a one-line answer in your FAQ. Neither feels good for a visitor at 9pm who just wants to know if you ship to Canada.

The setup above splits the difference. A small AI, kept strictly to your own docs, answers the easy questions in real time and shows where the answer came from. Anything past its limits becomes a clean handoff to a human, full transcript attached — no “please tell me more” loops. And the questions it can’t answer don’t disappear; they pile up in one review queue you spend ten minutes on each week, so month two’s assistant is noticeably better than month one’s.

The next four posts walk through each piece in turn — how a conversation starts and stays alive, how the answerer stays grounded in your docs, how a handoff to a human works without making the visitor repeat themselves, and how gaps become better answers. One diagram per post. A cost breakdown and a final engineering reference at the end.

PART 2 OF 7

MAY 1, 2026 PART 2 OF 7 · WEBSITE CHAT ASSISTANT SERIES ~4 MIN READ

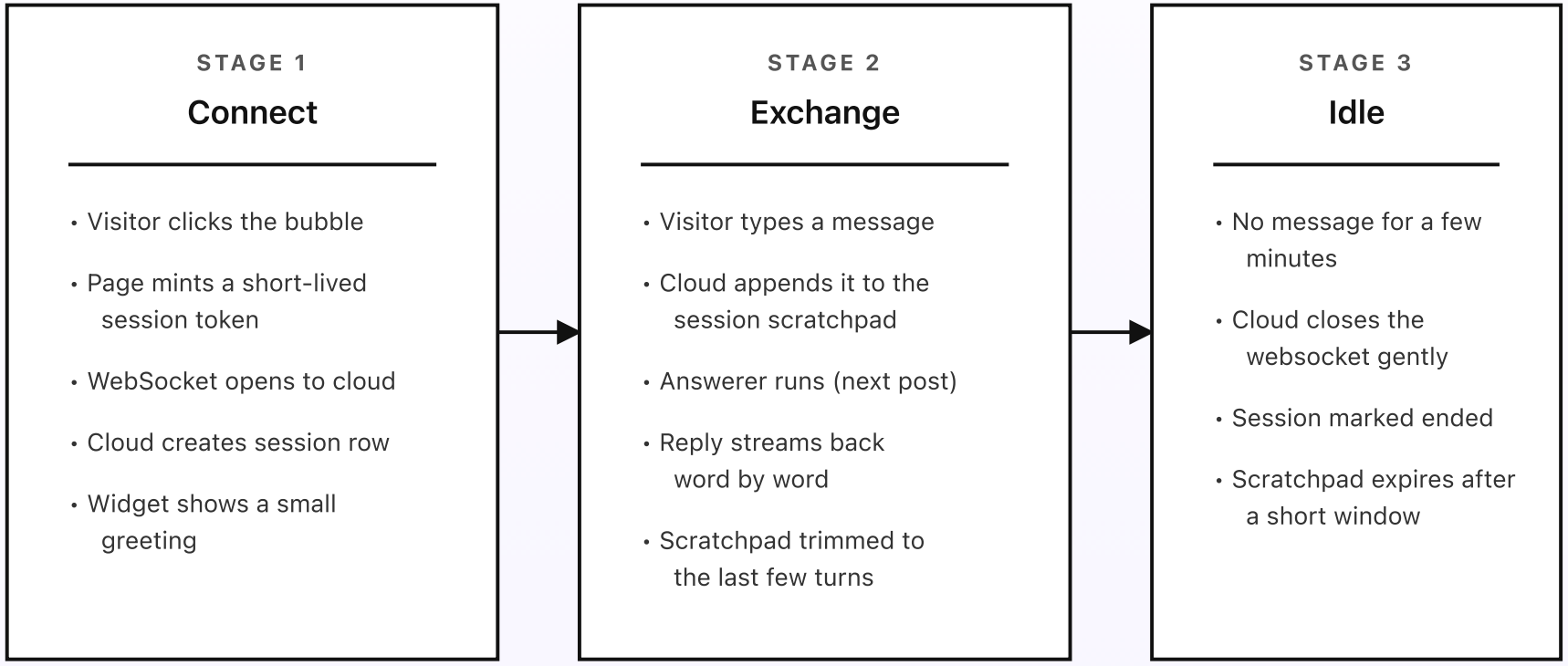
How a conversation starts and stays alive

A chat widget feels live, but underneath it's a careful little dance — the page loads a tiny script, the visitor clicks, a websocket opens, a scratchpad gets minted, and somewhere in between the cloud has to decide how much it's willing to remember. Here's the lifecycle of a single conversation, end to end.

KEY TAKEAWAYS

- Three stages per conversation: connect (mint a short-lived token, open WSS, write a session row), exchange (append turn, run answerer, stream reply), idle (close gently, expire the scratchpad).
- The scratchpad is short-term memory only — capped at the last few turns so “how long does it take?” can resolve to “shipping to Canada” without history growing unbounded.
- Each tab gets its own session and scratchpad; refresh drops the websocket and the new page mints a fresh one. No cross-page identity by default.
- Mid-reply disconnect is handled: the cloud finishes generating, stores it on the ended session, and shows it as the last turn when the visitor reconnects.
- Long-term memory is opt-in only — for signed-in customers who want their order history available, bound to an authenticated `customer_id` at `$connect`.

Three moments in a conversation



Short-term memory only — the system does not remember the visitor next week.

No long-lived profile unless the visitor explicitly signs in.

Fig 2. Connect, exchange, idle. A websocket and a small scratchpad — that's the whole shape.

Why a websocket and not a plain request

A chat widget is the one place on a website where a visitor expects words to appear in real time, the way a human typing would. Plain request/response works for a single question, but it falls apart on the second turn — the visitor sees a frozen UI while the cloud thinks, and follow-up questions feel laggy. A websocket makes the connection feel alive: messages stream back word by word as they're generated, and the visitor can type again without waiting for a full reply to arrive.

It also keeps each message cheap to send. Once the connection is open, each turn is a few bytes back and forth — no fresh login or security handshake every time. That's the difference between a 200ms chat and a 1.5-second chat. Visitors feel the second one as "laggy" even if they can't name what's wrong.

The session scratchpad

Every conversation has a small scratchpad — a short list of recent turns that the answerer reads before each reply. It exists so a visitor doesn't have to re-establish context on every message: if turn one was "do you ship to Canada?" and turn two is "how long does it take?", the scratchpad lets the answerer connect "it" to "shipping to Canada."

The rules for the scratchpad are deliberately strict:

- **Short.** Only the last few turns. Long histories make replies slower, more expensive, and weirdly off-topic as the AI reads things from earlier in the conversation that aren't relevant anymore.
- **Trimmed automatically.** Once the scratchpad reaches its limit, the oldest turn falls off. No manual cleanup, no growth without bound.
- **Session-scoped.** The scratchpad belongs to one websocket session. When the session ends, it's gone. The next time the visitor opens the widget, it's a fresh start.
- **Expires quickly.** Even before the session ends, idle scratchpads time out. Nothing the visitor typed lingers in storage longer than the conversation lasted.

The reason for all this restraint is simple: "memory" in a chat assistant is the source of half its problems. Long-term memory raises privacy questions (what did you store about that visitor?), causes surprise behaviour (the assistant brings up something from three weeks ago), and runs up the bill (every turn pays to re-read a long history). Short-term memory, kept to one session, avoids all three. If you want longer memory — for a signed-in customer who wants their order history available — that's a separate, opt-in feature you add on top.

What happens at the edges

Three edge cases are worth designing for from the start, because they're common and silent if mishandled:

- **The visitor refreshes the page.** The websocket drops; the widget reopens it on the new page; the new connection gets a fresh scratchpad. Treating "same visitor" across page loads adds complexity that almost no SMB needs.

- **The visitor opens two tabs.** Each tab gets its own session. They don't share a scratchpad. This is the simplest behaviour and what visitors expect — if they want to compare two threads, they expect them to stand alone.
- **The connection drops mid-reply.** The cloud finishes generating the reply, stores it on the (now-ended) session, and on next reconnect the widget shows it as the last turn. The visitor sees their answer the moment they come back online.

How this plugs into the next post

Everything in this post is plumbing — how a turn arrives at the answerer with the right context attached. The next post is about what the answerer actually does: how it searches your knowledge, how it requires a citation, and how it picks one of four moves on every visitor turn. The session and scratchpad are what let it focus on the current message without losing the thread.

PART 3 OF 7

MAY 1, 2026 PART 3 OF 7 · WEBSITE CHAT ASSISTANT SERIES ~5 MIN READ

How the assistant answers

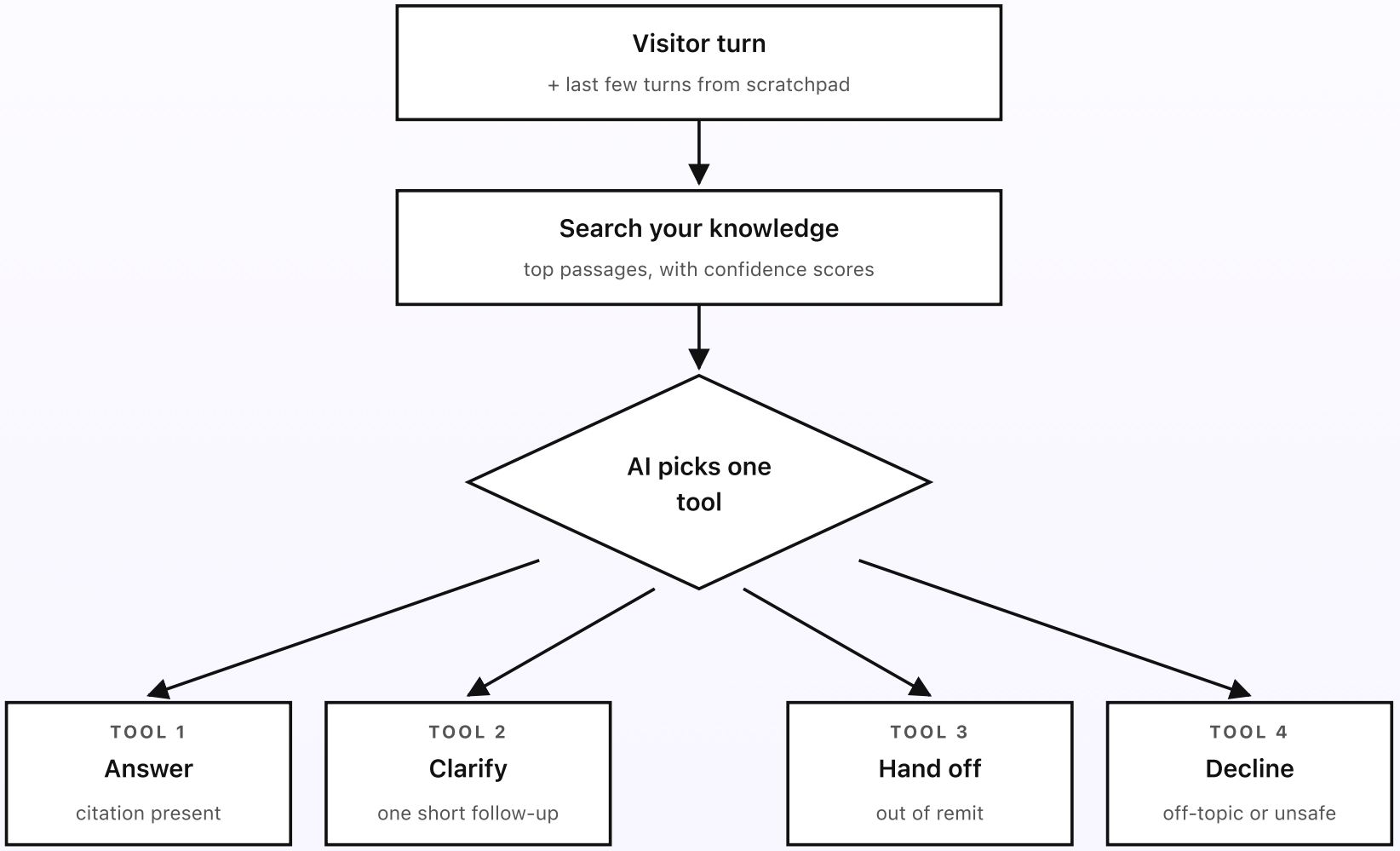
A visitor types a question. The assistant has a few seconds to do the right thing — and the "right thing" is rarely just "reply with words." Sometimes it's a clarifying question. Sometimes it's a quiet handoff. Sometimes it's "I don't know — and I won't guess." Here's how that decision gets made on every turn, and how a citation makes the difference between a real answer and a confident hallucination.

KEY TAKEAWAYS

- Search runs first, generation second. The current turn plus the scratchpad become a query against your Drive-backed knowledge base; only the retrieved passages are in scope for the reply.
- Strict tool_use, four tools per turn: `answer` (citation required), `clarify` (one short follow-up), `hand_off` (out of remit), `decline` (off-topic or unsafe).
- Citation is enforced twice: prompt forbids the answer tool without a supporting passage, and the runtime verifies the cited id was in the retrieved set before flushing.
- Below the confidence line, only clarify, hand off, and decline are allowed — the assistant leans toward asking or escalating when docs are thin, never inventing.
- If the model emits an answer with a citation outside the retrieved set, the runtime downgrades to `hand_off` — the safer-by-default failure mode.

Four tools, one pick per turn

Every visitor message ends in one of four outcomes. The assistant picks exactly one. There is no “maybe both” or “answer and also escalate” — ambiguity in the routing is what makes assistants feel chaotic.



No citation, no auto-answer. Below threshold routes to clarify or hand off — never to answer.

Fig 3. One tool per turn. The decision is the first thing the AI does — the words come second.

Search before generation

The first thing that happens on every turn is search, not generation. The current message and the last few turns from the scratchpad get turned into a query. The query goes against your knowledge folder — the help docs, FAQ, policies, and pricing pages you maintain in Drive. The result is a small list of passages that look most relevant to what the visitor just asked, each with a confidence score.

This order is deliberate. If the AI wrote first and searched second, it would lean on things from its training that aren't true about your business — old prices, generic shipping times, made-up return rules. By making the AI read your passages first, the only facts it has are the facts you wrote.

The search itself is small. It runs against a managed knowledge base that follows your Drive folder; an edit to a help doc shows up in search results within minutes, no deploy. You don't maintain an index, you don't run a search database, you don't schedule re-indexing — the managed piece handles all of that. What you do is keep your help docs accurate, the way you would for a new hire.

The four tools

Answer is the happy path. The retrieved passages cover the visitor's question with high confidence. The AI is asked to write a short reply that cites the source — one or two sentences from the passage, stitched into a sentence that reads

naturally. The reply streams back over the websocket so words appear within a second. A small “from: shipping policy” tag goes underneath, so the visitor can click through if they want the full version.

Clarify handles the case where the question is real but ambiguous. “Do you have any in stock?” without a product. “What time?” without a service. The AI is allowed to ask exactly one short follow-up — not three, not a form. After the visitor replies, the next turn flows through the same four tools again.

Hand off covers everything that’s on-topic but beyond what the docs can answer confidently. Refund disputes, custom orders, account-specific questions, anything where being wrong is more expensive than being slow. The AI tells the visitor a human will pick this up shortly; the next post is about how that handoff actually lands.

Decline is for the rest. Off-topic small talk, attempts to make the assistant write code or essays, hostile or unsafe asks. The reply is brief and polite (“I’m here to help with questions about [your business] — want me to grab a human if you have a different question?”) and the turn is logged for review.

Citation as a hard gate

The single most important rule — the one that separates a useful chat assistant from a confident liar — is this: **no citation, no answer.** If the AI can’t point to a passage from your docs that backs up what it’s about to say, the “answer” tool is off the table. The route falls through to clarify (if the question is real but your docs are thin) or hand off (if your docs don’t cover the topic at all).

The rule is enforced in two places. First, the AI is told to only use the answer tool when it has a supporting passage. Second, the system double-checks that the cited passage really was in the search results before sending the reply. If the AI tries to cite something it didn't actually retrieve, the system catches it and switches to hand off instead. Belt and braces — because "please don't make things up" is fine advice, but advice that isn't enforced fails when it matters.

Confidence and routing

Each retrieved passage comes with a score. Above a chosen line, the answer tool is allowed. Below it, only clarify, hand off, and decline are. This means the assistant leans toward asking or handing off when the docs are thin — which is exactly the lean you want. A visitor who gets a clarifying question feels heard. A visitor who gets a wrong answer feels lied to.

Picking that line is a slider, not a science. Set it cautious on day one (more handoffs, fewer answers) and loosen it as your gaps log fills up — the next post is about handoffs, and the one after that is about turning weekly gap reviews into better answers.

PART 4 OF 7

MAY 1, 2026 PART 4 OF 7 · WEBSITE CHAT ASSISTANT SERIES ~4 MIN READ

How a handoff to a human works

The handoff is the most important part of a chat assistant, and the part most assistants get wrong. The bad version makes the visitor repeat themselves to a human; the worse version drops the transcript on the floor; the worst version pretends to escalate and never does. Here's how to design a clean one.

KEY TAKEAWAYS

- Four steps in fixed order: tell the visitor with a realistic window, package the transcript, deliver to one destination, hold the session for follow-up turns.
- The visitor is told *before* the backend work starts — honest window ("within business hours," not "within an hour" at 9pm Friday) and human language, not "a ticket has been created."
- The package always carries five things: full transcript (no summary substitute), one-line AI-written summary, page URL, contact, and the reason for handoff (out of remit, low confidence, explicit request, or unsafe).
- One destination per business — inbox, Slack, or shared queue, never both. Two destinations means two people see it, both assume the other will reply, and neither does.
- The websocket stays open for a couple more turns to catch "actually never mind, I figured it out" — appended as a follow-up note so the human knows not to send a wasted reply.

What a handoff has to do, in order

From the moment the assistant picks the "hand off" tool, four things have to happen, in this order, before the visitor closes the tab:

STEP 1

Tell the visitor

A single short line, a realistic window, an inline ask for email or phone if needed.

STEP 2

Package the transcript

Full conversation + one-line summary + page URL + contact + reason for handoff.

STEP 3

Deliver to one place

Inbox, Slack, or shared queue — whichever the team already watches. One destination, never two.

STEP 4

Hold the rest of the session

WebSocket stays open for any extra context, then closes gracefully.

The visitor never repeats themselves to the human. The transcript is the ticket.

Fig 4. Four steps. The visitor experience is built around "tell first, deliver second" — the message lands before the queue does.

Step 1: tell the visitor first

The very first thing that happens, before any backend work, is that the assistant streams a short, plain line back to the visitor: *"Got it — I'll have a human follow up. We usually reply within business hours."* Two things matter here. The window is honest (not "within an hour" if it's 9pm Friday) and the language is human (not "a ticket has been created").

If the assistant doesn't already have a way to reach the visitor — no email captured earlier, no phone — this is the moment to ask, in the same chat. One question: *"What's the best email to follow up on?"* Not a form, not a popup. Just one line in the chat the visitor can reply to. The point is to make "getting in touch" feel like part of the conversation, not a fee for getting an answer.

Step 2: package the transcript

Once the visitor knows a human is coming, the cloud assembles a small payload. The contents are unglamorous but specific:

- **The full transcript** — every visitor turn and every assistant turn from this session. Not a summary, not the "important parts." The whole thing, so the human can read it the way the visitor experienced it.
- **A one-line summary** — the AI's last useful job. What did the visitor want? "Visitor asked about a refund on order #12345 placed two weeks ago, after we

couldn't find a matching policy." This is what the human reads in the notification preview.

- **Where they were** — the page URL the chat opened on. "On the pricing page" is meaningfully different from "on the contact page."
- **How to reach them** — the email or phone they shared, or a session reference if they're still on the page.
- **Why the handoff happened** — one of: out of remit, low confidence, explicit request from the visitor, or hostile/unsafe message. The team uses this to spot patterns.

Step 3: deliver to one place

The package goes to exactly one destination. The team's existing inbox, or Slack channel, or shared queue — whichever they're already watching. Not the inbox *and* Slack: in practice that means two people see it, both assume the other will reply, and neither does. Pick one place per business and route everything there.

The format matches the destination. To Slack, the summary becomes a one-line preview with a button to expand the full transcript. To email, the summary is the subject line and the transcript is the body. To a queue, it's a structured row. Same payload underneath, three packagings.

Step 4: hold the session for a moment

After step three, the websocket doesn't close immediately. The visitor often types one more thing — the order number they forgot to include, the email they meant to

send, “actually never mind, I figured it out.” The session stays open for a couple of turns; any new turns get appended to the transcript and forwarded as a follow-up note to the same destination. After that, the websocket closes and the session ends.

The “actually never mind” case is the underrated one. The system should let the visitor cancel cleanly — the assistant acknowledges, the team gets a small “visitor self-resolved” note appended to the original handoff, and the human knows not to send a wasted follow-up.

What this is not

This is on purpose *not* a full ticketing system. There’s no priority queue, no reply-time clock, no auto-assignment, no agent dashboard. For a small business, those are too much on day one and a chore every day after. The handoff is a thin pipe from the chat into your existing inbox; the inbox you already use becomes the “dashboard.” If you outgrow that, you can plug in a real ticketing tool later — but most businesses never need to.

PART 5 OF 7

MAY 1, 2026 PART 5 OF 7 · WEBSITE CHAT ASSISTANT SERIES ~4 MIN READ

How gaps become better answers

An assistant that can't answer a question isn't a failure — an assistant that can't answer the same question in *two months* is. Here's the small loop that turns every miss into a paragraph in your docs, and turns next month's identical question into a clean auto-reply.

KEY TAKEAWAYS

- Four-step clockwise loop: log every miss, group similar questions, write a paragraph, re-index automatically. Ten minutes a week.
- Every clarify, hand off, and non-hostile decline appends a row to `tbl-gaps` with the visitor turn, page URL, timestamp, and the closest passages found — no extra PII.
- A weekly EventBridge cron (`0 6 ? * MON *`) groups gaps by Titan Embeddings cosine similarity so “ship internationally?” and “order from the UK?” collapse into one row with a count.
- Writing standard: a sentence the assistant could quote word-for-word and have it sound right. No marketing voice, no hedging.
- Re-indexing is automatic — the Bedrock managed KB watches the Drive folder via `fn-drive-sync` and picks up new paragraphs in about ten minutes (5 to sync, 5 to index). No deploy.

The loop, in one diagram

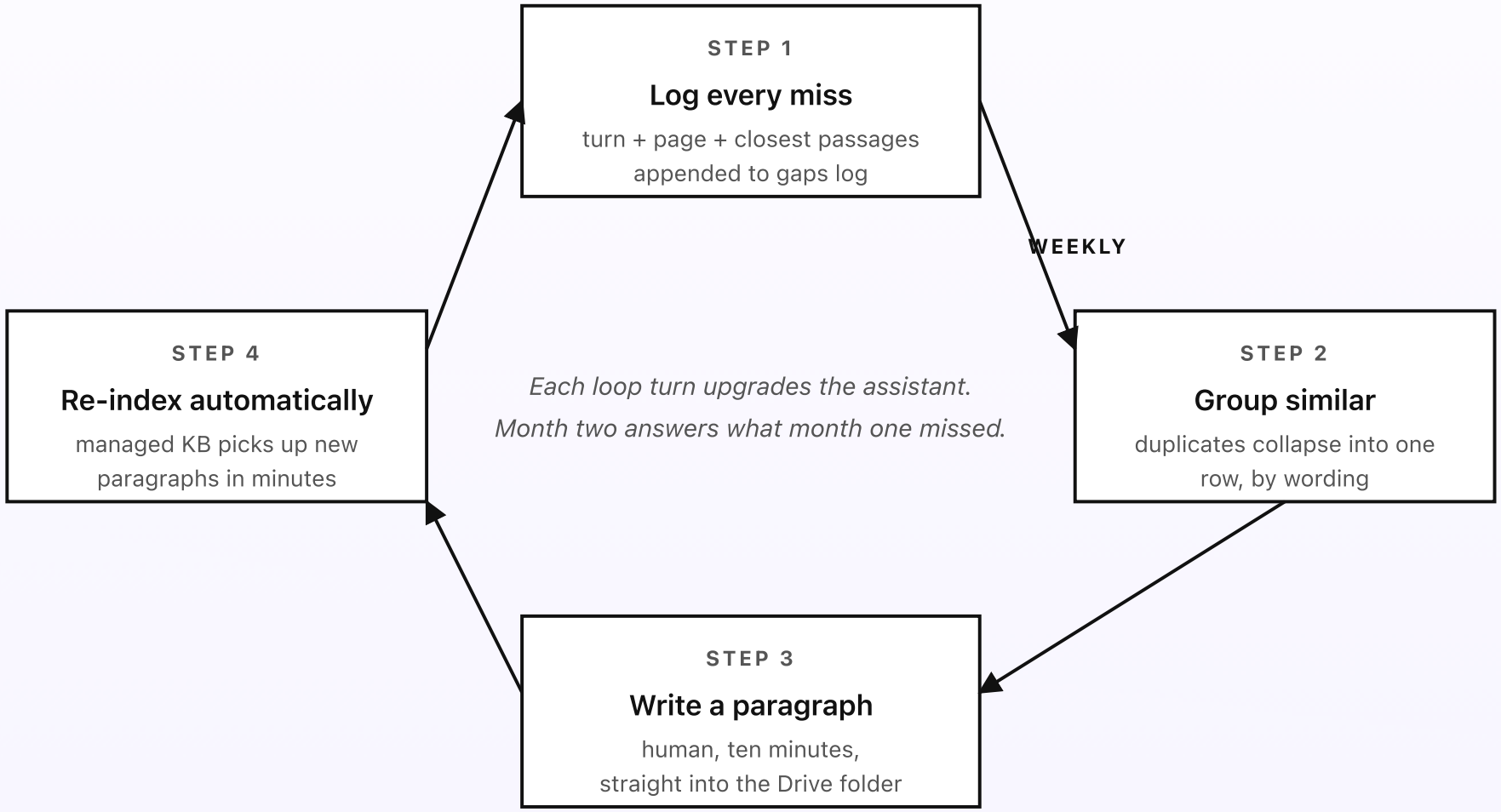

STEP 1
Log every miss
turn + page + closest passages appended to gaps log
WEEKLY
STEP 2
Group similar
duplicates collapse into one row, by wording
STEP 3
Write a paragraph
human, ten minutes, straight into the Drive folder
STEP 4
Re-index automatically
managed KB picks up new paragraphs in minutes
Each loop turn upgrades the assistant.
Month two answers what month one missed.

Fig 5. A small clockwise loop. Misses become groups, groups become paragraphs, paragraphs become better answers.

Step 1: log every miss, automatically

Every time the answerer picks “clarify,” “hand off,” or “decline” for a question that wasn’t hostile, the cloud appends a row to a small Drive doc — the gaps log. Each row is short: the visitor’s message, the page URL, the timestamp, and the closest passage the search did find (which might just be a near miss). Nothing else — no PII the visitor didn’t paste in themselves, no tracking IDs.

The reason this lives in Drive and not a database is simple: the team already lives in Drive. No dashboard to log in to, no permissions to set up. Open the doc, read this week’s misses, write the answers, save — that’s the whole workflow.

Step 2: group similar questions

Most weeks, the gaps log isn’t a long list of unique problems — it’s a short list of repeated ones, each asked by a few different visitors in slightly different words. “Do you ship internationally?”, “Can I order from the UK?”, “Where do you ship to?” are the same question dressed differently.

A small batch job runs once a week and groups gaps by similar meaning. Duplicates collapse into a single row with a count next to it: “international shipping (asked 7 times this week).” This is what makes ten minutes enough — you’re not writing seventy answers, you’re writing five or six, each of which covers ten or twenty visitor turns.

The grouping is rough, not perfect, and that's fine. The human reading the log still scans the list and might split or merge groups manually. The job's only job is to make the list short enough that opening it doesn't feel like a chore.

Step 3: write a paragraph

For each group, you write one short answer — two or three sentences — and add it to the right doc in your knowledge folder. International shipping goes into the shipping policy doc. A pricing question goes into the pricing FAQ. If no doc fits, make a small new one with a clear name ("Returns FAQ" instead of "misc-2.docx"); the search doesn't care about the file name, but you and the next person to edit will.

The writing standard is "a sentence the assistant could quote word-for-word and have it sound right." That usually means plain language, no marketing voice, and no hedging ("it really depends" is a bad answer; "we ship to the UK; orders take 7–10 business days" is a good one). If you can't write a clean answer because the policy itself is undecided, that's a real find — the gap log just surfaced it. Decide the policy, then write the paragraph.

Step 4: re-index, automatically

You don't deploy anything. The managed knowledge base watches the Drive folder; it sees the new paragraph within a few minutes and re-indexes the affected file. The next visitor who asks "do you ship to the UK?" gets a real answer with a citation pointing back at the paragraph you just wrote.

This last step is what makes the loop work. If updating the assistant needed a code change, a build, or a deploy — or even logging into a dashboard and re-uploading a file — the loop would die after the first week. Keeping the knowledge in Drive and the index on autopilot is what lets a non-engineer keep the assistant sharp.

What you'll notice in month two

Three things tend to happen, in roughly this order. First, the handoff volume drops — not because visitors are asking less, but because more of their questions now have grounded answers. Second, the gap log gets shorter and weirder — the bulk topics get covered, leaving only the genuine edge cases (which are usually fine to hand off; that's what humans are for). Third, the team starts noticing knowledge gaps in their own docs — the assistant is, in effect, auditing the FAQ by trying to use it. That last one is the most underrated benefit and the one that tends to surprise business owners who didn't expect a chat widget to also be a docs-quality tool.

PART 6 OF 7

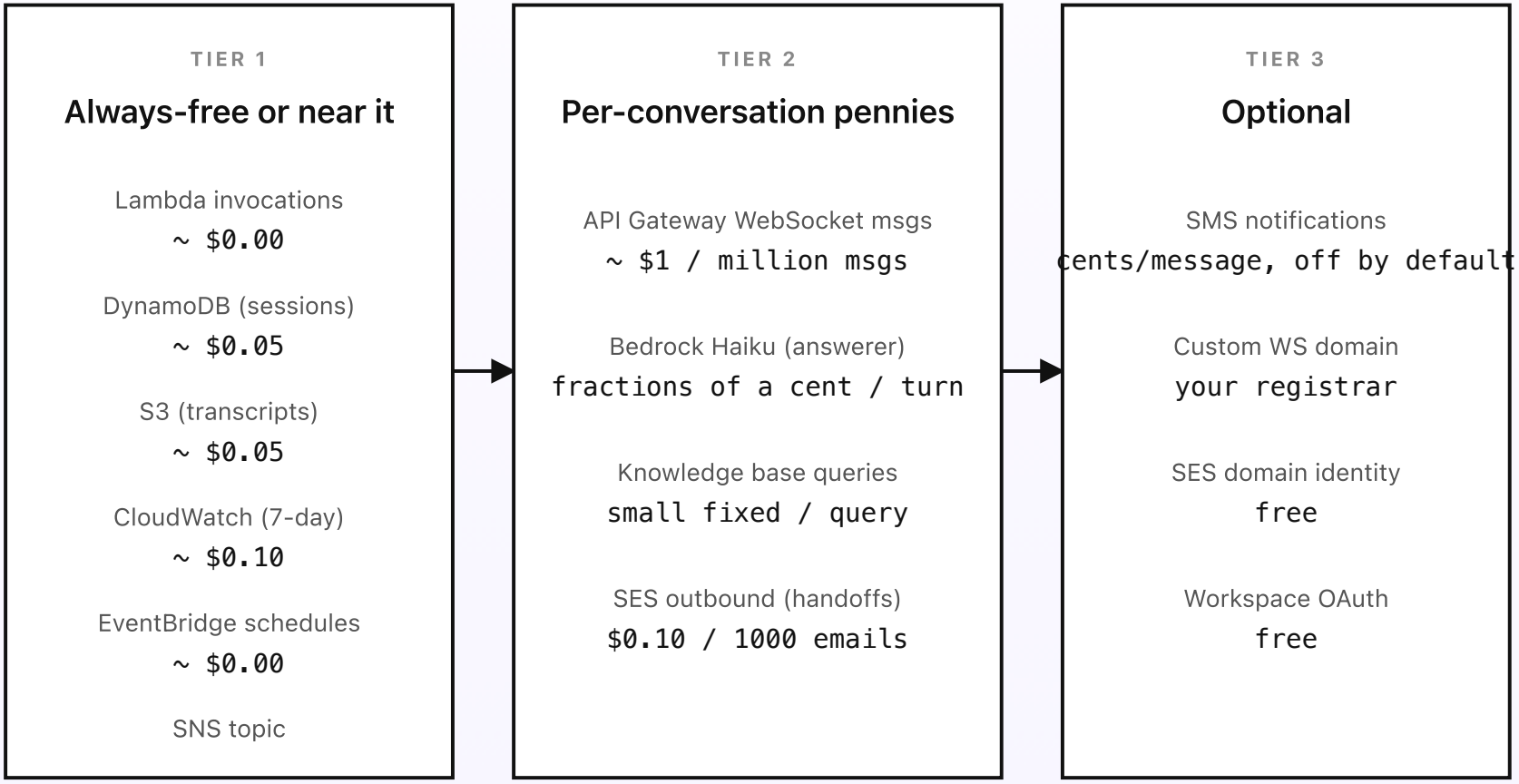
MAY 1, 2026 PART 6 OF 7 · WEBSITE CHAT ASSISTANT SERIES ~3 MIN READ

What the chat assistant costs

A coffee a month at typical SMB volume. The fixed cost is essentially zero — if the widget is quiet for a week, the bill for that week is too. The variable cost is dominated by Bedrock tokens for the answerer and the managed knowledge-base queries; everything else rounds to pennies.

KEY TAKEAWAYS

- Three cost tiers: always-free (Lambda, DynamoDB on-demand, S3, CloudWatch 7-day, EventBridge, SNS), per-conversation pennies (WebSocket messages, Bedrock Haiku tokens, KB queries, SES), and optional (SMS, custom WS domain).
- API Gateway WebSocket messages run about $1 per million; a hundred conversations is a fraction of a cent.
- Bedrock Claude Haiku 4.5 tokens dominate the variable bill — fractions of a cent per turn at small-model rates; a 5-turn conversation is still pennies.
- Vector store is Amazon S3 Vectors (managed) with no idle minimum — quiet sites pay nothing for retrieval, unlike OpenSearch Serverless or Aurora pgvector.
- A typical SMB at ~500 conversations a month lands under five dollars total, often under three. A $10 monthly AWS Budgets alarm catches anything strange.



~ 500 conversations a month → under five dollars total, often under three.

Fig 6. Three tiers of cost. The bill scales with how often the widget gets opened; the floor is nearly zero.

The fixed cost is essentially nothing

There is no per-seat licence and no minimum monthly fee. If your site has a quiet week, the AWS bill matches. Lambda, DynamoDB pay-per-request, S3, CloudWatch, EventBridge, and SNS all sit in or near the always-free tier at the volumes a small business chat sees. The biggest single line item in tier one is usually CloudWatch log storage, and even that you control by retaining seven days instead of forever.

The API Gateway WebSocket has a tiny per-connection-minute charge while a session is open. At a few minutes per conversation and a few hundred conversations a month, it rounds to under fifty cents.

The variable cost is per-conversation pennies

Three things scale with traffic:

- **WebSocket messages** — about $1 per million. A conversation is typically tens of messages (each visitor turn plus a streamed reply), so a hundred conversations is a fraction of a cent.
- **Bedrock Haiku tokens** — the answerer reads a short prompt and writes a short reply on each turn. At small-model rates, this lands at a fraction of a cent per turn. A 5-turn conversation is still pennies.

- **Knowledge-base queries** — one query per turn. The managed retrieval (S3 Vectors as the default vector store, since mid-2025) has a small per-query cost and a tiny token cost for the embedded passages. S3 Vectors has no idle minimum — you only pay when the widget actually runs a query — so a quiet site costs nothing. Together: still cents at hundreds of conversations.

Add it up: most small businesses end up between $1 and $5 a month total at a few hundred conversations. The widget pays for itself the first weekend it answers "do you ship to Canada?" without a human on call.

Three traps you're avoiding

- **Per-seat live-chat tools.** Most off-the-shelf chat-widget products charge $20–$80 per agent per month, regardless of volume. You're trading a flat per-seat bill for pay-per-use that mostly comes in pennies.
- **Running a vector store yourself.** The managed knowledge base avoids the "tiny database that costs $40 a month and you have to babysit" trap. With S3 Vectors as the default, even the managed option has no idle floor — quiet sites pay nothing for retrieval. You write Drive docs; the index follows.
- **Long scratchpads.** Memory that grows unbounded multiplies your token bill turn after turn. Trimming to the last few turns keeps cost flat.

When this stops being cheap

The math changes if traffic grows ten or a hundred times — thousands of conversations a month with long multi-turn threads. At that point Bedrock token

spend becomes the headline number, not a footnote. There are levers to pull: shorter scratchpads, fewer retrieved passages per turn, a small cache for repeat questions. Most small businesses never need them.

For everyone below that — and that's most small businesses — a $10 monthly AWS Budget alarm catches anything strange before it becomes a surprise on the credit card.

In plain words

The fixed bill is nearly zero. The variable bill is cents per conversation, dominated by tokens. A typical small-business setup runs at coffee-money for the whole month. Set a budget alarm that fits your expected volume and the bill can't surprise you.

PART 7 OF 7

MAY 1, 2026 PART 7 OF 7 · WEBSITE CHAT ASSISTANT SERIES ~5 MIN READ

Engineering reference: the chat assistant architecture

Same system as the rest of the series, drawn purely for engineers. Service names, resource identifiers, region, Bedrock model IDs, Knowledge Base wiring, and the actual flow operations — everything you'd need to recreate this in your own AWS account.

KEY TAKEAWAYS · VERIFIED MAY 2026

- Single AWS account in `ap-southeast-1` (Singapore); Bedrock via Global cross-Region inference.
- Five subsystems: Build & Deploy, Knowledge Sync, Conversation gateway (Lambda Function URL + API Gateway WebSocket), Answerer (RetrieveAndGenerateStream + strict tool_use), Handoff & learning.
- Models: `global.anthropic.claude-haiku-4-5-20251001-v1:0` + `amazon.titan-embed-text-v2:0`; vector store is S3 Vectors (managed by Bedrock).
- WebSocket routes `$connect` / `$default` / `$disconnect` on `wss-chat`; replies stream via `ApiGatewayManagementApi.PostToConnection`; widget mints a short-lived JWT from `fn-mint-token` to keep long-lived secrets out of the browser.
- Citation enforced at runtime: if the model emits `answer` with a citation outside the retrieved set, the runtime downgrades to `hand_off`.

Posts 1–6 walk through the system in plain language. This page is the dense version — nothing softened, just the architecture as you'd sketch it on a whiteboard during a design review.

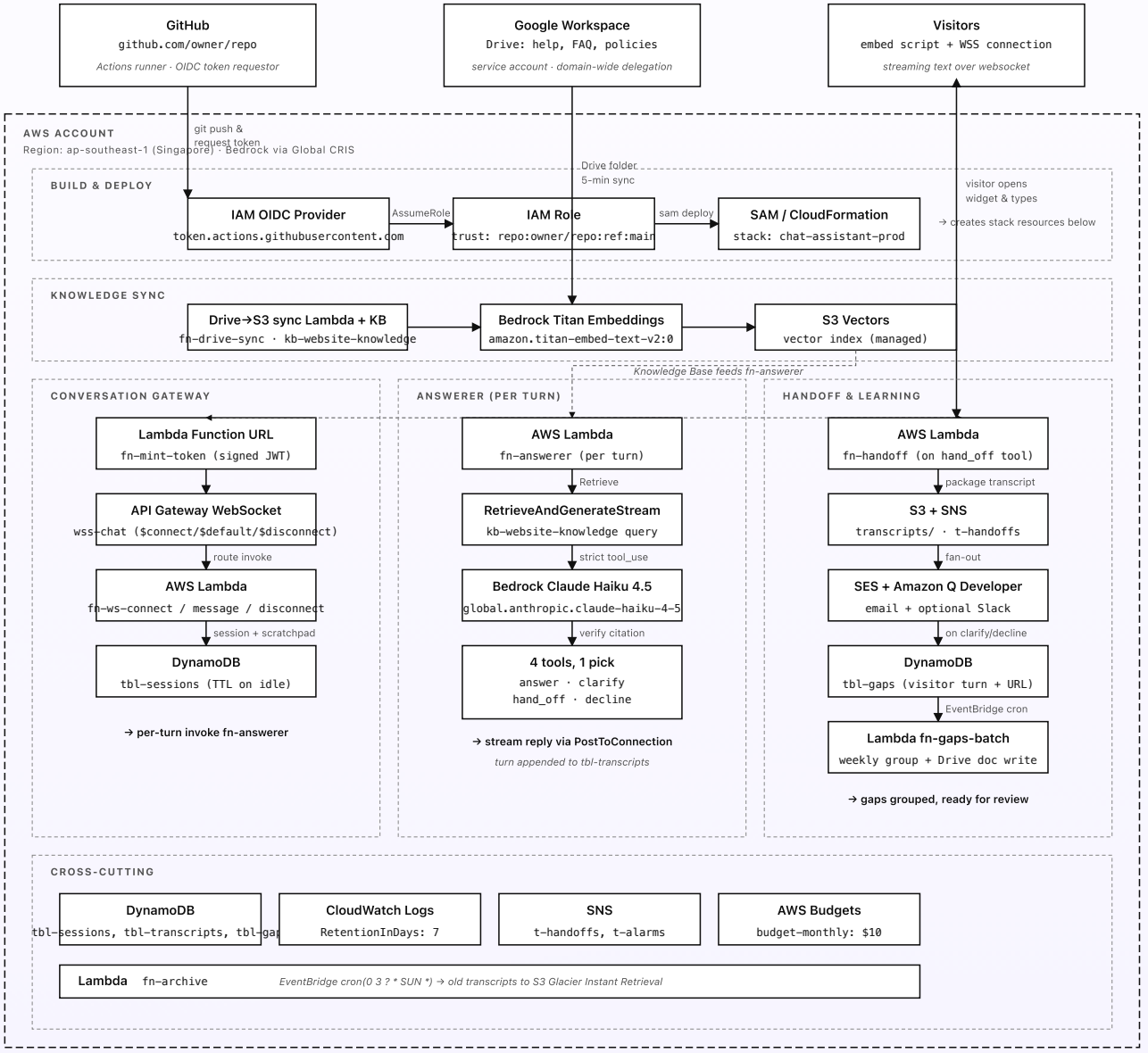

GitHub
github.com/owner/repo
Actions runner · OIDC token requestor
Google Workspace
Drive: help, FAQ, policies
service account · domain-wide delegation
Visitors
embed script + WSS connection
streaming text over websocket
AWS ACCOUNT
Region: ap-southeast-1 (Singapore) · Bedrock via Global CRIS
git push & request token
Drive folder 5-min sync
visitor opens widget & types
BUILD & DEPLOY
IAM OIDC Provider
token.actions.githubusercontent.com
AssumeRole
IAM Role
trust: repo:owner/repo:ref:main
sam deploy
SAM / CloudFormation
stack: chat-assistant-prod
→ creates stack resources below
KNOWLEDGE SYNC
Drive→S3 sync Lambda + KB
fn-drive-sync · kb-website-knowledge
Bedrock Titan Embeddings
amazon.titan-embed-text-v2:0
S3 Vectors
vector index (managed)
Knowledge Base feeds fn-answerer
CONVERSATION GATEWAY
Lambda Function URL
fn-mint-token (signed JWT)
API Gateway WebSocket
wss-chat ($connect/$default/$disconnect)
route invoke
AWS Lambda
fn-ws-connect / message / disconnect
session + scratchpad
DynamoDB
tbl-sessions (TTL on idle)
→ per-turn invoke fn-answerer
ANSWERER (PER TURN)
AWS Lambda
fn-answerer (per turn)
Retrieve
RetrieveAndGenerateStream
kb-website-knowledge query
strict tool_use
Bedrock Claude Haiku 4.5
global.anthropic.claude-haiku-4-5
verify citation
4 tools, 1 pick
answer · clarify
hand_off · decline
→ stream reply via PostToConnection
turn appended to tbl-transcripts
HANDOFF & LEARNING
AWS Lambda
fn-handoff (on hand_off tool)
package transcript
S3 + SNS
transcripts/ · t-handoffs
fan-out
SES + Amazon Q Developer
email + optional Slack
on clarify/decline
DynamoDB
tbl-gaps (visitor turn + URL)
EventBridge cron
Lambda fn-gaps-batch
weekly group + Drive doc write
→ gaps grouped, ready for review
CROSS-CUTTING
DynamoDB
tbl-sessions, tbl-transcripts, tbl-gaps
CloudWatch Logs
RetentionInDays: 7
SNS
t-handoffs, t-alarms
AWS Budgets
budget-monthly: $10
Lambda fn-archive
EventBridge cron(0 3 ? * SUN *) → old transcripts to S3 Glacier Instant Retrieval

Fig 7. Full architecture, ap-southeast-1. White boxes = AWS resources; dashed AWS container; dashed grey boxes = subsystem groupings; dashed grey arrows = config feed and side branches.

Read this top-down, then column-by-column

Top row is the three external surfaces. Below it, the AWS account contains five subsystems: Build & Deploy across the top, then Knowledge Sync, then three runtime columns (Conversation gateway, Answerer, Handoff & learning), with a Cross-cutting strip at the bottom. A visitor opens the widget, the page calls `fn-mint-token` for a short-lived JWT, and connects to `wss-chat`. The `$connect` route writes a session row; each `$default` message invokes `fn-ws-message` which appends to the session scratchpad and invokes `fn-answerer`. The answerer issues a Bedrock `RetrieveAndGenerateStream` against `kb-website-knowledge` with strict tool_use over four tools (`answer`, `clarify`, `hand_off`, `decline`), streams the reply back via `PostToConnection`, and writes the turn into `tbl-transcripts`. `hand_off` invokes `fn-handoff`; `clarify` and `decline` append to `tbl-gaps` for weekly review.

Naming conventions used in the diagram

- **Lambda functions:** `fn-<purpose>` — `fn-mint-token`, `fn-ws-connect`, `fn-ws-message`, `fn-ws-disconnect`, `fn-answerer`, `fn-handoff`, `fn-gaps-batch`, `fn-drive-sync`, `fn-archive`.
- **Lambda runtimes:** Python 3.13 for the answerer, handoff, gaps batch, drive sync, and archive functions; Node.js 22.x is fine for `fn-mint-token` and the

WebSocket route handlers if you prefer JS. Both runtimes are current LTS-equivalents for Lambda as of 2026-05.

- **DynamoDB tables:** `tbl-sessions` (partition key `connection_id`, with a `scratchpad` list trimmed to the last few turns and a TTL on idle), `tbl-transcripts` (partition key `session_id`, sort key `turn_index`), `tbl-gaps` (partition key `week_iso`, sort key `created_at#turn_id` with the visitor turn, page URL, closest-passage scores).
- **SNS topics:** `t-handoffs` for human-handoff fan-out (email, optional Slack), `t-alarms` for general failures.
- **S3 layout:** single bucket `chat-assistant-data` with prefixes `transcripts/{date}/`, `archive/`.
- **Knowledge Base:** `kb-website-knowledge`, a Bedrock managed Knowledge Base with an **S3 connector** pointed at the synced help/policies prefix. Bedrock KBs do not have a native Drive connector as of 2026-05, so a small `fn-drive-sync` Lambda mirrors the Drive folder to S3 on a 5-minute schedule. Embeddings model is `amazon.titan-embed-text-v2:0`; vector store is **Amazon S3 Vectors** (the cheapest quick-create option since mid-2025 — zero idle cost — provisioned and managed by Bedrock when you create the KB). OpenSearch Serverless and Aurora PostgreSQL Serverless remain valid alternatives if you outgrow S3 Vectors' query throughput.

Region, model access, websocket details, and Drive auth

Everything runs in `ap-southeast-1` (Singapore). Bedrock model invocations use the **Global cross-Region inference** profile (`global.` prefix on model IDs) — data at rest stays in Singapore; inference may route to other regions for capacity, billed at on-demand Singapore rates.

The widget mints its session token from `fn-mint-token` rather than authenticating directly against API Gateway; the JWT is short-lived (a few minutes) and is checked in `fn-ws-connect` via a Lambda authorizer. This keeps long-lived secrets out of browsers entirely. Streaming replies use `ApiGatewayManagementApi.PostToConnection` with chunked writes — the answerer flushes partial responses every few tokens so the visitor sees words appear within a second.

Google Drive authentication uses a service account with **domain-wide delegation** over a single scope: `https://www.googleapis.com/auth/drive.readonly` on the help-docs folder only. The credential lives in AWS Secrets Manager. The `fn-drive-sync` Lambda runs on a 5-minute EventBridge schedule, pulls any changed docs from Drive, writes them to `chat-assistant-data/kb-source/`, and lets the Bedrock KB's S3 connector index from there. Editing a doc and saving propagates within ~10 minutes (5 to sync + 5 to index); manual re-sync is one CLI call to `StartIngestionJob`.

The answerer uses **strict tool_use**: four tool definitions (`answer`, `clarify`, `hand_off`, `decline`) with required parameter schemas. The `answer` tool requires a `citation_id` parameter referencing one of the retrieved passages by id; the runtime validates the citation against the retrieved set before allowing `PostToConnection` to flush. If the model emits an `answer` with a citation that

wasn't in the retrieved set, the runtime downgrades to `hand_off` — the safer-by-default failure mode.

What's deliberately not on the diagram

- IAM policy details — per-Lambda execution roles are minimal (one bucket prefix, one or two tables, a single Bedrock KB ID, `InvokeModel` on one model, `execute-api:ManageConnections` on one API).
- Per-business knowledge layout — a flat Drive folder is fine for the first few months; subdivide by topic (`shipping/`, `returns/`, `pricing/`) once it grows past a couple of dozen docs, so writers know where new paragraphs go.
- X-Ray tracing — on for `fn-answerer` and `fn-handoff`, sampling 100% during tuning, 10% in steady state.
- **Bedrock Guardrails** — managed contextual grounding (numeric grounding + relevance scores), PII redaction, prompt-attack/jailbreak filters, and the newer **Automated Reasoning checks** (formal-logic policy validation, GA in 2025). The custom citation-verification step in `fn-answerer` is roughly the contextual-grounding idea hand-rolled; turning on Guardrails moves the threshold into console configuration and adds PII redaction and prompt-attack defence on every model call. Worth enabling once thresholds are stable.
- **Long-lived visitor identity** — for logged-in customers who want their order history available, swap `connection_id` for an authenticated `customer_id` at `$connect` and bind the scratchpad to that identity. Keep it opt-in.
- **Multi-tenant variant** — if running this on behalf of multiple SMBs, namespace the KB and tables per tenant and inject `tenant_id` into every record. The

architecture doesn't change shape; the IDs do.

- **Slack two-way handoff** — the diagram fans out to Slack as a notification only. A bidirectional Slack-to-visitor reply path (agent types in Slack, visitor sees it in the widget) is an additional Lambda + Slack Events Subscription; off the default diagram to keep the per-message cost in the always-free band.

> **IF YOU'RE RECREATING THIS**
>
> Start with Build & Deploy alone (a single Lambda, no triggers). Once `git push` reliably updates an empty stack, wire up `fn-drive-sync` with one help doc and confirm the doc lands in S3 within five minutes. Create the Bedrock Knowledge Base over that S3 prefix and confirm a one-shot `RetrieveAndGenerateStream` call returns a passage. Then the WebSocket API with stub `$connect` / `$default` / `$disconnect` handlers that just echo back. Then the real `fn-answerer` with strict tool_use and citation verification (this is the part most worth integration-testing — intentionally try to make the model cite a passage outside the retrieved set and confirm the runtime downgrades to `hand_off`). Then the handoff fan-out and the gaps log. Cross-cutting (audit, logs, alarms, budget, archive) goes in from day one.