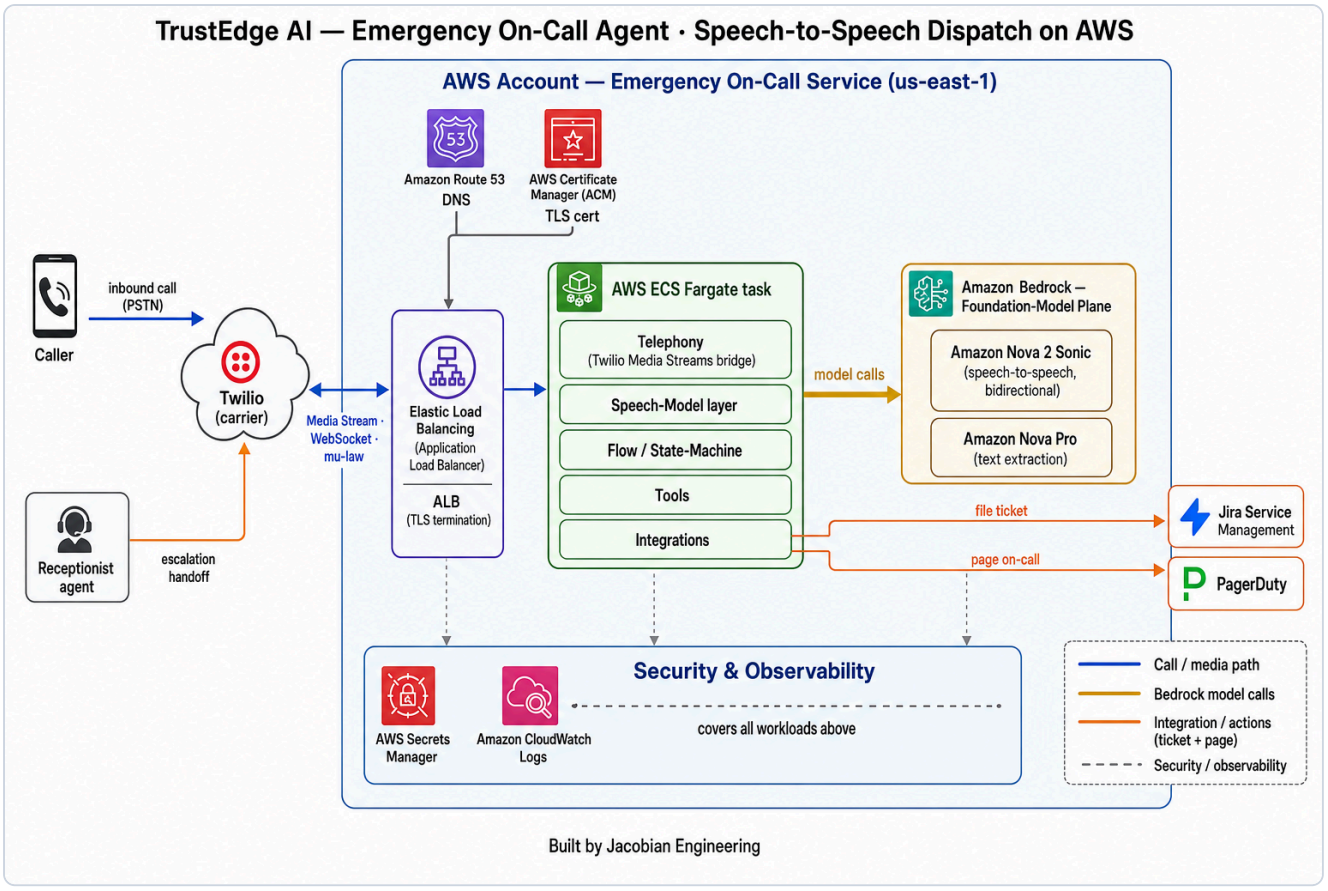
TrustEdge AI: An Autonomous Speech-to-Speech On-Call Dispatcher



Architecture at a Glance

1. Executive Summary

Jaiven is an autonomous, speech-to-speech AI emergency on-call dispatcher. It answers TrustEdge AI's own after-hours emergency line, holds a natural spoken conversation with a stressed caller, extracts a complete and accurate incident report, files a correctly-routed and deduplicated ticket in the service-management platform, pages the on-call engineer, attaches the call recording to the ticket, and closes the call — with no human in the dispatch loop. The caller hears a calm, consistent voice; the engineer receives a structured, deduplicated page; and every action behind the call is logged and auditable.

TrustEdge built Jaiven for its own line first, as a research and reference implementation — dogfooding before productizing. TrustEdge AI is the AI services division of Jacobian Engineering, an employee-owned IT-security, compliance, and

cloud-services firm with two decades of trust-critical infrastructure work behind it, serving regulated sectors including healthcare and financial services. The after-hours emergency intake is exactly the kind of problem the division productizes its internal systems against: high-stakes, mostly quiet, occasionally critical, and unforgiving when handled badly. Rather than ship a voice-AI product built on a slide deck, TrustEdge ran the agent on its own emergency number, found the failures that only surface under real telephone traffic, and engineered them out. The result is the flagship reference implementation for how the division builds production voice AI.

The thesis the build proved, repeatedly, is this: a generative model is an excellent conversational surface and an unreliable process controller. Production reliability comes from moving every decision, every action, and every guarantee out of the model and into deterministic application code — while letting the model do the one thing it is genuinely good at: talking to a human. That sentence is not a slogan; it is the architecture. The speech model owns voice, a text model owns extraction, and deterministic code owns state, decisions, actions, and guarantees. Everything else in this document follows from that division of labor.

> *"We put this on our own emergency line before we offered it to anyone else, because an emergency dispatcher is the worst possible place to discover that your AI improvises. The model is a wonderful talker and a terrible dispatcher, and the only way you learn that is to let it answer a real call from a real person whose system is down at two in the morning. We did, it improvised, and we redesigned the whole thing around the assumption that it always will." — Erik D. Jones, Lead Architect & CEO, TrustEdge AI*

This case study is the engineering story behind that decision.

[[TOC]]

---

2. The Problem

Managed IT and security firms live and die by their after-hours response. When a client's production system or clinical application fails at 2 a.m., the gap between the call landing and the right engineer being paged is measured in minutes that matter. During business hours a human can triage; after hours the firm needs a dispatcher that is awake at all times, never inconsistent, and never fatigued. The cost of staffing a 24/7 human dispatcher for a call volume that is mostly quiet but occasionally critical is difficult to justify — and yet the cost of *missing* a critical call, a clinic's electronic health record system down overnight, is unacceptable. That is the operational squeeze the emergency intake sits inside.

The conventional options are all poor. A voicemail box is a black hole until morning; nobody hears it, and the incident ages while it waits. An answering service mangles

technical detail, transcribes "the EHR cluster is unreachable" into something an engineer cannot act on, and adds a human relay that introduces both latency and error. A rotating human duty-phone burns out the on-call staff it depends on, and it is precisely the staff the firm most needs rested when the page finally comes. Each of these fails the moment that matters most, and each fails in a way the business cannot see until it is too late to fix.

The person on the other end of an emergency call is rarely calm. Their system is down, they are under pressure, and they need three things fast: to feel that a competent party is now handling it, to receive a ticket number that proves the incident is logged, and to know that an engineer is being summoned. They must not be forced through a touch-tone menu, asked to repeat themselves, or left wondering whether anything actually happened. The agent's mandate is therefore narrow and deep: be a warm, efficient dispatcher that **collects the incident accurately, files it correctly, and escalates it reliably** — and does nothing else. It explicitly does not troubleshoot, diagnose, or advise. That boundary is not a limitation; it is a safety property, and enforcing it turned out to be one of the central engineering problems of the build.

The naive implementation — "point a conversational LLM at a phone line and tell it to take incident reports" — fails in production in ways that only appear at the boundary between a generative model and a real telephone network. A speech model trained on the breadth of human conversation will *improvise*. Told to follow an eleven-step script, it will skip steps, reorder them, troubleshoot the caller's problem because that is what its training data is full of, and occasionally claim it cannot perform actions it can in fact perform. Telephony audio is low-fidelity — 8 kHz, full of codec artifacts and silence patterns that confuse turn-detection. Cloud-platform behavior diverges between a developer's laptop and a production container in subtle, authorization-shaped ways that pass every local test and fail the first real call. And most fundamentally, the model cannot be trusted to reliably terminate a call, page an engineer, or stay on-script — and yet those are exactly the guarantees the business requires. The rest of this document is about how each of those problems was solved by taking the corresponding responsibility away from the model.

---

3. TrustEdge's Approach

Jaiven is a single service that fronts a telephone number and bridges it to a cloud speech model, and the whole design rests on three bets about who owns what. The temptation when building voice AI is to hand the model everything — let it hear the caller, decide what to ask, decide when it has enough, decide to file the ticket and page the engineer, and decide when to hang up — and trust that a sufficiently capable model will behave. That design produces a compelling demo and an unreliable service. TrustEdge rejected it from the first redesign in favor of a strict separation of concerns that runs through every layer of the system.

Bet one: the speech model owns voice, and only voice. The conversation itself is carried by a speech-to-speech foundation model. It receives raw audio in and emits raw audio out over a single bidirectional stream, handling speech recognition, reasoning, and speech synthesis in one pass. That is what it is genuinely good at — fluent, low-latency, natural-sounding spoken delivery and robust recognition over a noisy phone line. For an emergency caller, where conversational latency directly shapes how "handled" they feel, a single-pass speech model was the right primary surface over a recognize-then-think-then-synthesize cascade. The model's job is to talk and to listen. It is not asked to do anything else.

Bet two: a separate text model owns extraction. A speech model is superb at conversation and poor at emitting clean, structured per-turn data; it is not a structured-output model. So the system runs a second, text-only model purely to extract the fields of an incident report from the running transcript. This is a classic two-pass design, and it is the backbone of the system's reliability: the speech model handles the human, the text model handles the data, and neither is asked to do the other's job.

Bet three: deterministic code owns state, decisions, actions, and guarantees. What to ask next, whether the report is complete, when to file, what severity to assign, whether to page, how to route, whether a filing succeeded, and when to hang up — none of these is left to a model. They live in a server-side state machine and a finite catalog of server-executed actions. The model proposes nothing consequential; the server decides and acts. This is the bet that turns a demo into a service, and the remaining chapters are largely an account of building it out and the live-call failures that forced it.

> *"The model is the easy part. People think the hard work of a voice agent is the conversation, and it is the most visible part, but the speech model does that almost for free. The hard work is everything the model must not be allowed to decide — and the discipline to keep it out. We use one model to talk and a second model to turn what was said into structured data, and we let neither one near a decision. That split is ninety percent of why the thing is reliable." — Erik D. Jones, Lead Architect & CEO, TrustEdge AI*

---

4. Architecture Overview

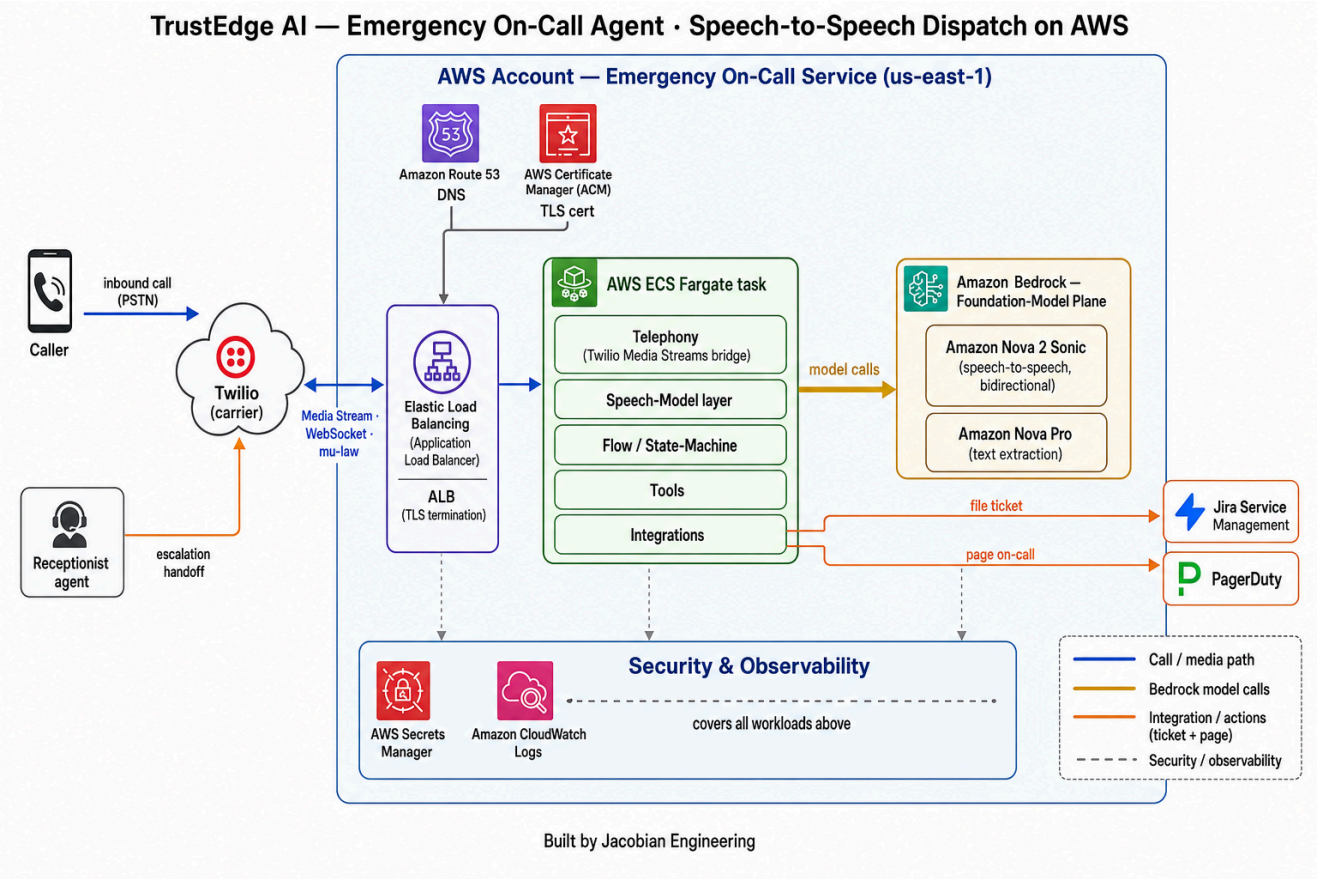


Figure 1 — System architecture: Twilio Media Streams bridged into a single ECS Fargate task, Amazon Nova 2 Sonic and Nova Pro on Amazon Bedrock, with deterministic filing to Jira Service Management and paging through PagerDuty.

The entire service runs as a single containerized process on a serverless container platform. Jaiven is deployed on **Amazon ECS Fargate** as one long-lived task, fronted by a dedicated **Application Load Balancer** that terminates TLS for both the HTTPS webhooks and the secure WebSocket media stream. The TLS certificate is issued and DNS-validated through **AWS Certificate Manager**, and a stable hostname is pointed at the load balancer through an **Amazon Route 53** hosted zone. The choice of a long-lived container rather than serverless functions is deliberate: the speech model uses a persistent bidirectional stream for the duration of a call, which does not fit a request/response function model. Because each WebSocket is pinned to a single task instance, a rolling deployment will drop calls in flight — an accepted operational constraint, mitigated by deploying during quiet windows, since the agent's traffic is by nature sparse and bursty.

The telephone side is carried by Twilio, end to end. An inbound call to the emergency line invokes the service's call webhook over HTTPS; the service responds with **TwiML** that instructs Twilio to play a greeting and then open a **Twilio Media Streams** connection — a bidirectional WebSocket carrying base64-encoded mu-law telephony audio in both directions. Twilio is also the carrier-control surface: the service uses Twilio's REST API to redirect (transfer) calls, to terminate them, and to control

recording. Twilio is the only thing that may legitimately drive the service, which makes its request signature the system's security perimeter — a point §10 returns to.

The conversation runs on Amazon Bedrock, across two models. Voice is carried by **Amazon Nova 2 Sonic**, the speech-to-speech foundation model, over a Bedrock bidirectional streaming connection at 8 kHz, 16-bit mono LPCM — a rate chosen to match Twilio's native telephony codec so the inbound path requires no resampling. Structured extraction runs on **Amazon Nova Pro**, a text model invoked through Bedrock's synchronous text API after each caller turn. Both models are reached through Bedrock, which makes Bedrock the single governed plane for every foundation-model call the service makes.

The code is organized into five cleanly separated layers, each with a single responsibility. The **telephony layer** owns the webhook handlers, the TwiML that instructs Twilio, signature validation, the real-time media bridge, recording control, and the hang-up backstop. The **speech-model layer** owns the bidirectional streaming client to Nova 2 Sonic, the event-protocol factory, and the session manager that holds the live stream. The **flow layer** sits centrally and owns the finite state machine, the immutable incident draft, the Nova Pro extraction client, and the orchestrator that ties them together. The **tools layer** owns the discrete, server-executed actions and the pure policy functions behind them. The **integrations layer** owns minimal REST clients for the ticketing platform, the paging platform, and Twilio. The service itself is built with **FastAPI** and served by **uvicorn**.

The deployment posture is infrastructure-as-code and keyless CI/CD. The stack is described in **AWS CDK** (Python) and shipped through **GitHub Actions**, which authenticates to AWS via **OIDC** — short-lived federated credentials rather than long-lived access keys. Secrets for the ticketing, paging, and carrier integrations live in **AWS Secrets Manager** and are injected at runtime; no secret is baked into the image or the source. The container image is held in **Amazon ECR**, and every spoken turn, state transition, and action result is logged to **Amazon CloudWatch Logs**. The greeting the caller first hears is rendered offline by **Amazon Polly** and is not a runtime dependency — covered in §7. Notably, there is no workflow-orchestration middleware in the path: the service calls Twilio, Bedrock, the ticketing platform, and the paging platform directly, by design.

For all that infrastructure, what the caller actually meets is a single, consistent persona. The architecture exists to make one thing true: that the voice on the line is always the same calm dispatcher, and that the dispatcher's behavior is bounded by design rather than by mood.

> *"We gave the agent a name — Jaiven — because the person calling at two in the morning is talking to someone, and that someone has to be the same calm, competent voice every single time. Jaiven's whole job is to be the unflappable dispatcher you wish answered every emergency line: it takes your name, it takes*

> *the incident, it tells you the engineer is being paged and hands you a ticket number — and it does not troubleshoot, it does not improvise, it does not editorialize. Naming it helped us hold that line. Jaiven is a dispatcher, not a chatbot, and everything it does or refuses to do follows from that one fact." — Erik D. Jones, Lead Architect & CEO, TrustEdge AI*

5. The Two-Model Split: Voice vs. Extraction

This is the decision that makes the rest of the system possible, so it is worth dwelling on. A speech-to-speech model and a text extraction model are good at opposite things, and the system's reliability comes from never confusing the two. Nova 2 Sonic is a conversational surface: it recognizes a stressed caller over a noisy 8 kHz line, reasons about what was said, and synthesizes a natural spoken reply, all in a single low-latency pass. What it is not is a structured-output engine. Asking a speech model to also emit clean, schema-valid per-turn JSON — caller name, company, summary, severity, callback number, confirmation flags — is asking it to do a job it was not built for, and it shows: the structure drifts, fields blur, and the very act of producing structured output degrades the conversational quality that was the reason to use a speech model in the first place.

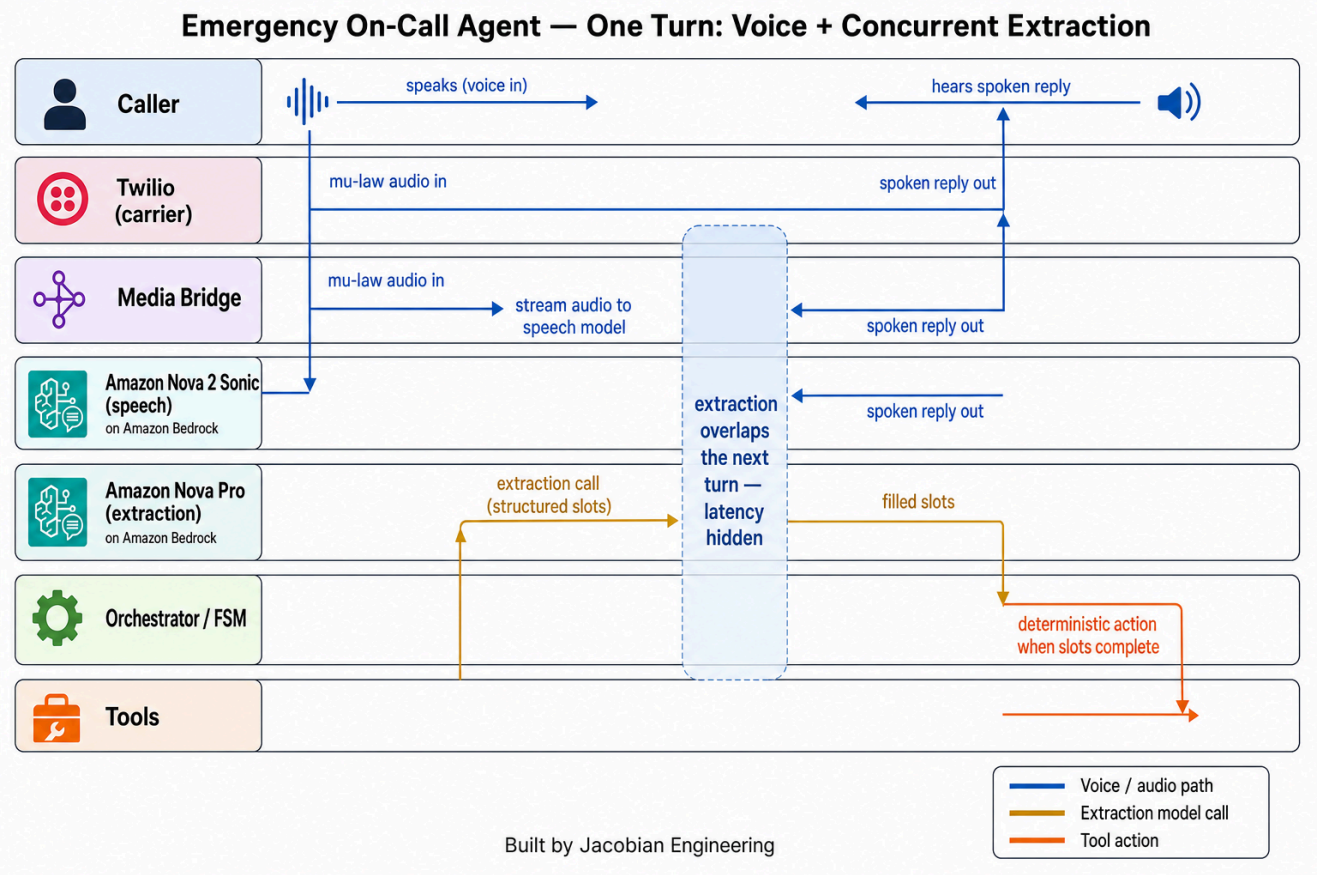


Figure 2 — A single conversational turn. The extraction pass on Nova Pro runs concurrently with the next spoken turn, so its latency is almost entirely hidden.

So extraction is a separate concern handled by a separate model. After each caller utterance — surfaced as a transcript from the speech model's recognition — the orchestrator runs **Amazon Nova Pro**, a text LLM, over the growing transcript. Nova Pro returns a JSON object of incident slots: the caller's name, the company they mentioned, a one-line summary of the problem, an inferred severity, a callback number, a confirmation flag, a "wants a human" flag, and a few optional fields. The text model is given exactly one job — turn an unstructured transcript into a structured draft — and it does that job reliably precisely because it is the only job it has.

The extraction pass runs concurrently, so its latency is hidden. Nova Pro is invoked in a background thread the moment a caller turn is recognized, and it runs while the speech model is already engaged with the next spoken turn. By the time the orchestrator needs the merged draft to decide what to do next, the extraction has typically already returned. The caller never waits on it. This is the first of several places in the system where a multi-second cost is deliberately overlapped with something the caller is already doing, so that the agent feels responsive rather than computational. The shape of the design is simple and load-bearing: **the speech model handles the human, in real time; the text model handles the data, in the background; and neither blocks the other.**

---

6. Conversation Control: From Prompt to a Server-Side State Machine

The first conversational design trusted the prompt, and it failed on live calls. The initial build gave the speech model a long, carefully numbered system prompt — an eleven-step intake script — plus a set of tools, and let the model run the conversation: deciding when to verify the caller's company, when to file the incident, when to end the call. Two live calls were enough to prove that a long, rigid system prompt does not control a generative speech model. It skipped company verification. It troubleshot the caller's problem despite explicit, emphatic instructions not to. And it did so *regardless* of how forcefully the prompt was worded or how low the inference temperature was set. The prompt was being delivered correctly; that was never the issue. The issue was architectural. A generative model with no application-level state machine is free to wander, and its training distribution pulls it toward rich, helpful, troubleshooting conversation rather than a terse intake script.

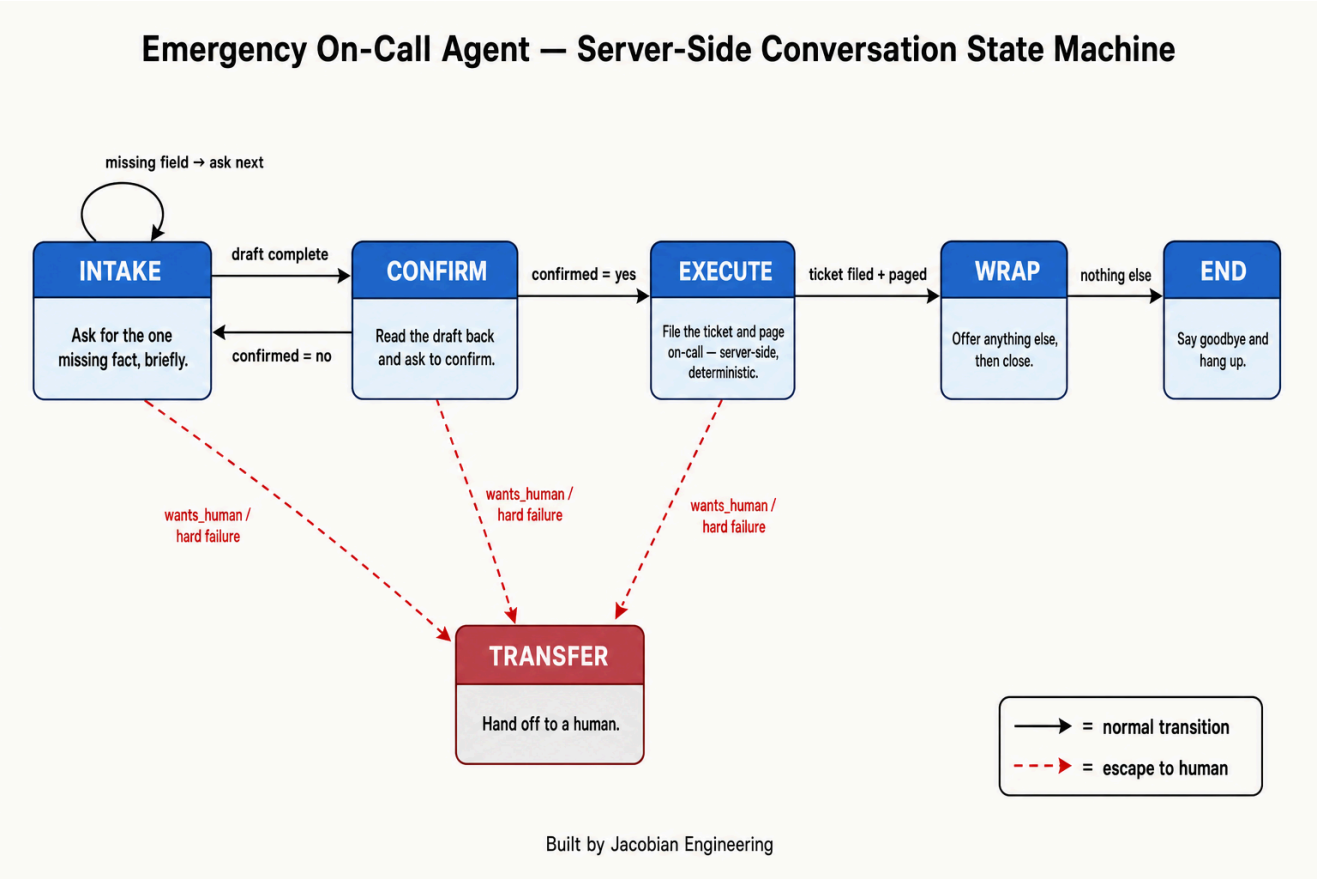


Figure 3 — The server-owned state machine. Each state injects exactly one sentence of intent into the model; the model never holds the script.

The fix was to rebuild the conversation around a server-side finite state machine that owns all state, decisions, and actions. In the new design the speech model's role collapses to two things: recognize the caller's speech, and say exactly the one sentence it is told to say next. It has no tools, and it makes no decisions. The machine itself is small and explicit. **INTAKE** is a slot-filling loop that asks only for the next missing required field. **CONFIRM** reads the assembled report back in a single sentence and asks for a yes/no once all required fields are present. **EXECUTE** files the ticket and pages, with no model involvement at all. **WRAP** narrates the ticket number and paging result and asks if there is anything else. **END** is a polite close and hang-up. **TRANSFER** is reachable from any state when the caller asks for a human, is unrecognized in a way that warrants escalation, or when an action hard-fails. State transitions are pure functions of the current state and the immutable incident draft, with explicit guards: a "wants a human" flag routes to TRANSFER, a complete draft advances INTAKE to CONFIRM, an affirmative confirmation advances to EXECUTE, and a negative one returns to INTAKE to re-collect.

The incident draft is an immutable, fill-only record, and that immutability prevents a whole class of bug. Required fields gate confirmation; optional fields never block. Merge semantics are fill-only: once a slot has a value, a later extraction pass can add information but can never blank it. This matters because telephony

transcripts are noisy — a later, garbled turn must never be allowed to erase a callback number that was captured cleanly three turns earlier. Each merge produces a new draft rather than mutating the old one, so the conversation's history is a sequence of monotonically more-complete records, never a value that flickers.

The subtle part is how the state machine steers the speech model, because the obvious mechanism is forbidden by the protocol. The natural approach — inject a fresh system instruction before each turn — is impossible: the speech model's streaming protocol permits only a *single system content block per session*, and injecting more would violate it. So the orchestrator steers the model with **interactive user-role text turns** instead. Before each agent turn it sends a short, imperative, state-specific instruction into the stream as if it were a user message — for example, *"Ask only which company they are with, in one short sentence; do not discuss the problem yet,"* or, at confirmation, *"Read this back in one short sentence and ask them to confirm yes or no: [summary], callback [number]."* The model treats each directive as the thing to respond to, and because the directive is re-asserted on every single turn, the model never gets the chance to decide what to ask. It cannot drift into troubleshooting or skip verification, because it is never given the latitude to choose what comes next. The intelligence of the conversation — what to ask, in what order, when enough is enough — lives in deterministic code; the model supplies only fluent delivery and recognition.

Both control modes were kept behind a feature flag, deliberately. The prompt-only mode was not deleted; it remains selectable, which makes the two strategies A/B-comparable on the same phone number without a rebuild and preserves the prompt-only path for future agents where strict scripting matters less. The default for the emergency line is the deterministic state-machine mode, for the reliability reasons above.

> *"We stopped asking the model to follow a script. That is the whole shift. A script assumes the model will remember step four after it has improvised through step three, and it will not — not reliably, not at any temperature, not with any amount of bold text in the prompt. So we took the script out of the prompt and put it in a state machine, and now the model is handed exactly one sentence of intent per turn, freshly, every turn, forever. It cannot wander off a script we never gave it." — Erik D. Jones, Lead Architect & CEO, TrustEdge AI*

---

7. The Telephony & Audio Pipeline

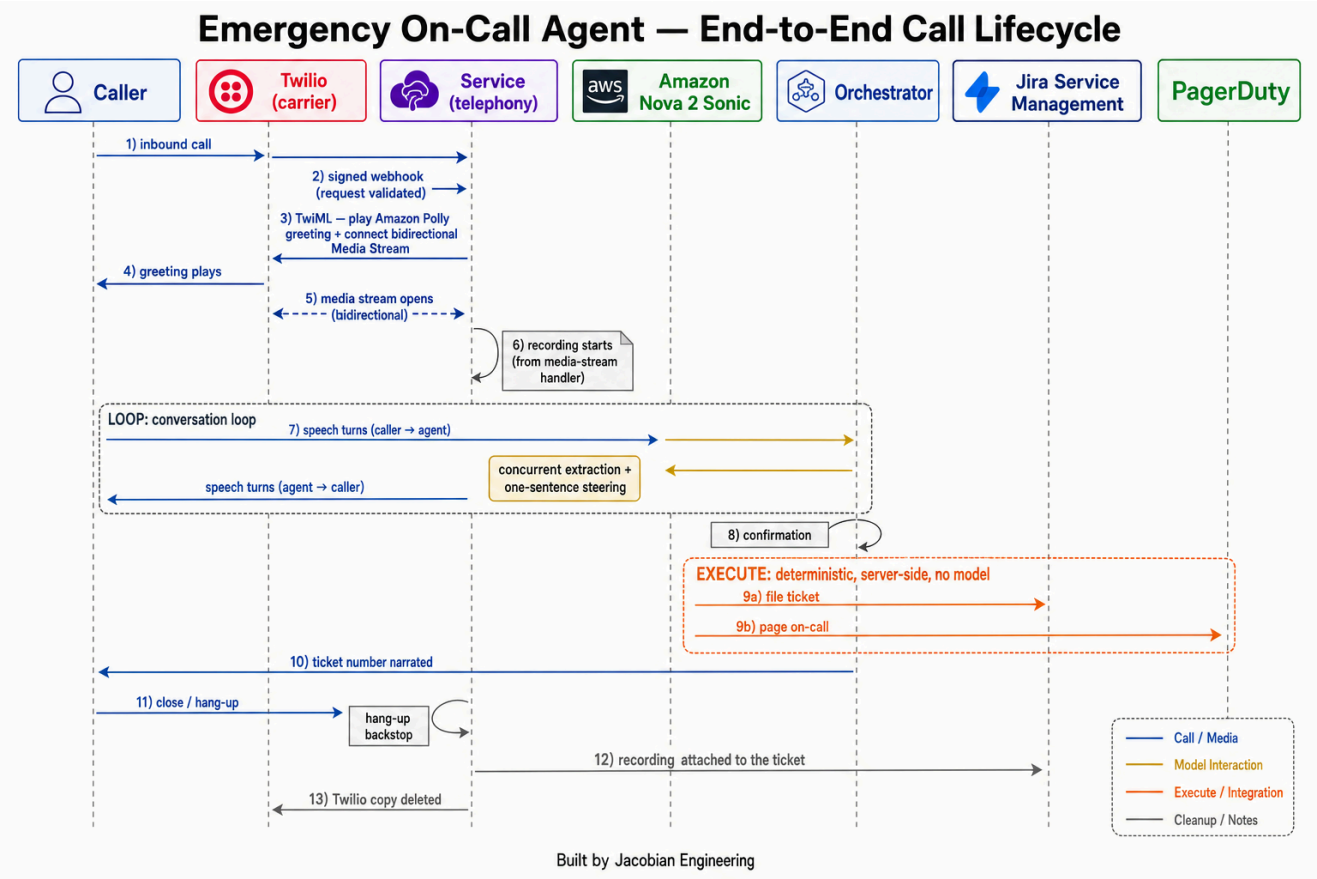


Figure 4 — The end-to-end call. The EXECUTE block — filing the ticket and paging the engineer — is deterministic and server-side, with no model in the loop.

The greeting is deterministic by design, and that single choice structurally deletes a hard problem. When a call arrives, Twilio invokes the service's call webhook, which returns TwiML instructing Twilio to play a pre-rendered greeting file and then connect a bidirectional media stream. The greeting is a static audio asset, synthesized offline by **Amazon Polly** in the "Matthew" neural voice — the same voice the speech model is configured to use — so the caller hears one consistent identity from the very first word. It includes the recording-disclosure line. Playing this greeting deterministically, before the model is ever in the loop, solved a real problem elegantly. Early on the team tried to make the model "speak first" by injecting a text trigger to prompt its opening line, and live tests returned no audio at all: a speech-to-speech model generates audio *in response to audio*, not in response to a text nudge. Rather than fight that, the design sidesteps it. Twilio plays the greeting, and the model joins only after it finishes — by which point the caller's first words, usually their name, are real audio the model can respond to naturally. The speak-first problem was not patched; it was structurally deleted.

The media stream is bidirectional, which sounds obvious and was in fact an inherited bug. The TwiML uses a connect-and-stream verb so audio flows both ways. An earlier version of the codebase used a one-way fork, which is precisely why audio the model produced was never heard — a foundational defect that no amount of

conversational tuning could have fixed, because the model's voice had nowhere to go. Once connected, Twilio streams audio frames over the WebSocket as base64-encoded mu-law payloads, and the media bridge runs a tight transcoding loop in both directions: inbound, it decodes base64, converts mu-law to 16-bit PCM, and enqueues for the model; outbound, it takes the model's PCM audio, converts to mu-law, and enqueues for Twilio. Because both ends run at 8 kHz, no resampling is required. A small portability detail surfaced here too: the standard-library audio module the transcoding relied on was removed in the project's Python version, so the build pins a maintained backport to restore it — the kind of dependency archaeology that real deployments require.

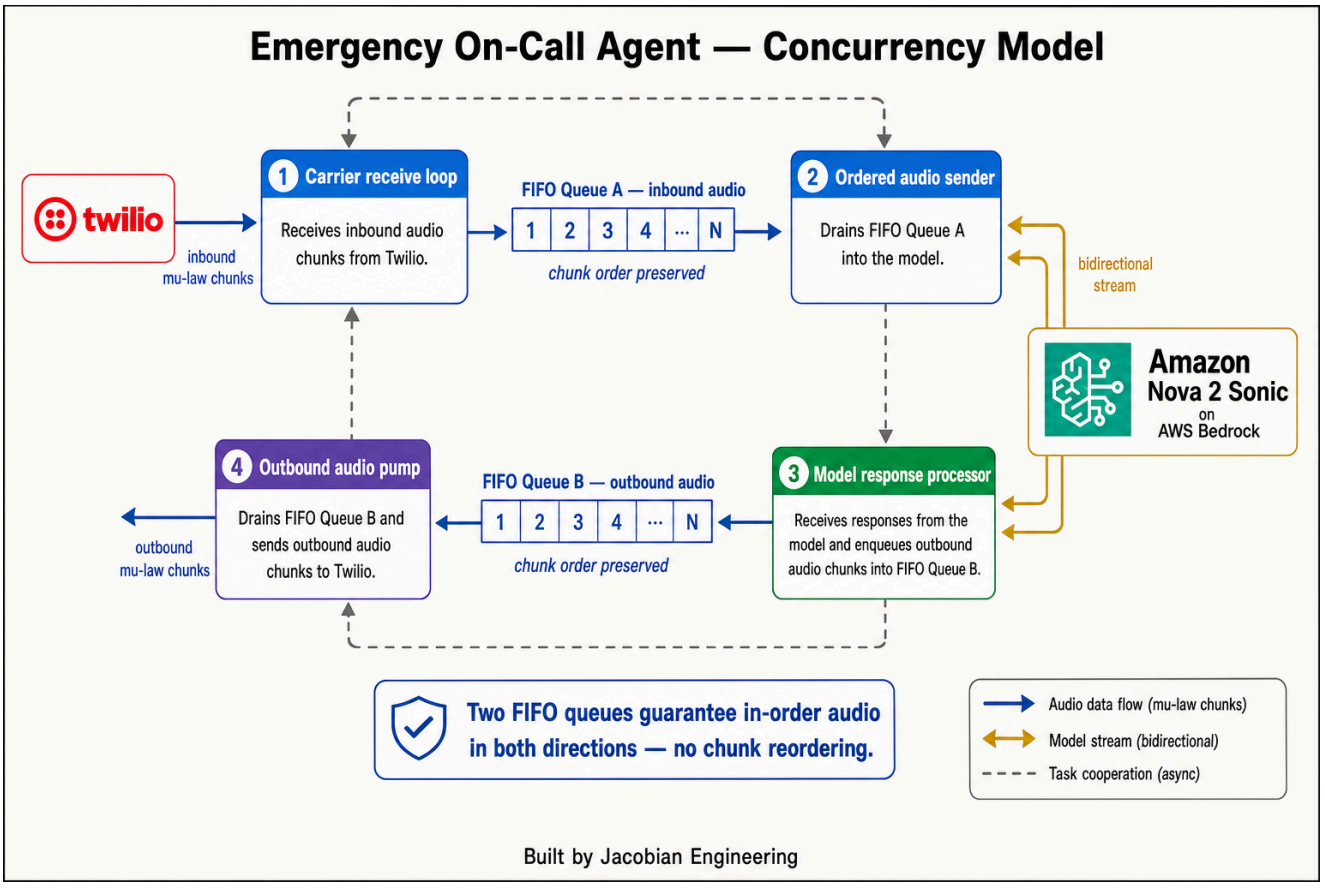


Figure 5 — Four cooperating asynchronous tasks and two FIFO audio queues guarantee chunk ordering in both directions — audio is intolerant of reordering.

Barge-in is essential on an emergency line, and getting it right requires two actions, not one. Stressed callers interrupt, and an agent that talks over them is the opposite of reassuring. The speech model signals when it detects the caller talking over it, emitting an interruption event — but acting on that event correctly takes two distinct steps, both of which an earlier version omitted. The service must **flush its own outbound audio queue** so the agent stops generating mid-sentence, *and* it must **send Twilio a buffer-clear** so audio already handed to the carrier is dropped before it plays. Flushing only the local queue leaves the carrier still playing a buffered sentence; clearing only the carrier buffer leaves the local queue refilling it. With both in place, the agent yields the floor immediately when the caller speaks. Under the hood, all of this

rides on a small set of cooperating asynchronous tasks per call, with inbound and outbound audio each draining a single ordered FIFO queue rather than fire-and-forget per-chunk tasks — because audio is intolerant of reordering, and an early "spawn a task per chunk" approach did not guarantee ordering.

Recording surfaced a classic platform-timing race worth remembering. Calls are recorded dual-channel — caller and agent on separate tracks — for the audit trail and for attachment to the resulting ticket. The first implementation started recording in the call webhook, the very first handler Twilio invokes, and Twilio rejected it: at that instant the call is not yet in progress — it is still awaiting the TwiML response — and a recording cannot be attached to a call in that state. The fix was to start recording from the media-stream handler, after the stream's start frame, at which point the call is unambiguously live and audio is flowing. The lesson generalizes to a rule the rest of the build leaned on: side effects that depend on a call's state must be issued at the lifecycle point where that state actually holds, not at the earliest convenient handler.

When the recording finishes, it is attached to the ticket and the carrier copy is deleted — a deliberate data-minimization control. Twilio calls a recording-status webhook on completion; the service looks up which ticket the call produced (through a short-lived in-memory map from call identifier to ticket key), downloads the audio, attaches it to the ticket as an internal artifact, and then deletes the Twilio-side copy. The recording, which may contain sensitive information, lives as the system of record on the ticket and nowhere else. For a firm whose calls may carry regulated healthcare or financial detail, that single-home-plus-disclosure handling is a concrete control, not a posture.

---

8. Actions: a Finite, Server-Executed Tool Surface

Whatever control mode the system runs in, the actions it can take are a small, fixed catalog, each executed by the server rather than the model. Even in the legacy prompt mode, the model only *requests* an action; the server decides whether and how to perform it. There are five actions, and there is no general-purpose, shell, or arbitrary-HTTP capability behind them. The model cannot act outside these lanes because there is no mechanism for it to do so.

Verify-customer fuzzy-matches a spoken company name against the customer roster. It uses a substring-aware similarity match that handles a caller saying a short form of a longer registered name — the kind of thing a human dispatcher does without thinking and a naive exact-match would fail. In state-machine mode this runs entirely server-side; the model never sees the logic and never decides the outcome.

File-incident is the heart of the system, and it is pure deterministic code from end to end. Given the collected fields, it validates the required ones — re-asking through the flow if any are missing — and **defaults the callback number to the**

caller's network-provided number when none was given. It resolves the correct ticketing queue and request type for that specific customer. It plays a short "one moment while I create your ticket" audio clip so the caller is not met with silence during multi-second API calls. It creates the ticket, sets its priority and organization tags, pages the on-call engineer for verified customers, and computes a **deduplication key** so that repeat calls about the same outage within the hour collapse to a single page rather than waking an engineer twice. It posts a full internal comment carrying the entire captured record, registers the call-to-ticket mapping so the recording can be attached later, and **arms the hang-up backstop**. It returns a narration string for the agent to read back and a set of structured flags describing any partial failure. Not one of these steps is a model decision.

The remaining three actions round out the surface. Transfer-to-human redirects the live call to a human line through Twilio's REST API. **End-call** returns immediately so the agent can speak its goodbye, then hangs up after a short flush delay so the goodbye audio fully drains before the line drops. **Add-note** appends a post-filing detail to the ticket if the caller volunteers more information after the incident is already logged.

The caller's network-provided number is server-captured and model-immutable, and that property is load-bearing. Twilio supplies the calling number; the server captures it and injects it into every ticket as an unforgeable field. The model is told to read it back as the default callback, but it cannot fabricate or alter the recorded value — the model's output is never the source of truth for an identity-bearing field. The same discipline governs severity: it is **assigned by the server from a rubric** (a site-down is critical, multiple users affected is high, and so on), never a label the caller picks, and deliberately never read aloud as jargon.

Partial-failure handling is explicit, deterministic, and never the model's call. If only the page fails, the ticket number is still narrated to the caller with a caveat. If only the ticket fails, the page result is narrated. If *both* fail, the system routes the caller to a human rather than pretend success. The model is never in a position to decide to skip a page or fake a ticket; those outcomes are computed in pure policy functions and narrated, not invented. This is the concrete shape of "every guarantee in code": the caller is told the truth about what happened because what happened was determined by deterministic logic, not by a confident generator.

---

9. Integrations

Jaiven integrates with two external systems of record through minimal, purpose-built REST clients — and the integration work, not the model, was where most of the engineering went. Tickets are created in **Jira Service Management Cloud**, and pages are dispatched through **PagerDuty**. Both are reached directly from the service; there is no orchestration middleware in between.

Ticketing is Jira Service Management Cloud, and tickets are created as Service Desk requests, not generic issues. The integration files each incident through JSM's Service Desk request API, routed to the per-customer service desk and request type that the file-incident action resolved. Each ticket is priority-tagged and organization-tagged, and it carries an internal comment with the full captured record — caller, network number, callback, scope, severity, and paging status — written in Atlassian's document format. The call recording is attached to that same ticket as an internal file, making the ticket the single system of record for the call's audio. Getting the JSM service account's permissions right was its own small saga: an early build had to treat a permission error on internal comments as success until the account was granted the correct service-desk role, and the workaround was removed once the permissions were corrected. That detail is representative of the whole engagement's central lesson: production AI agents are mostly integration and permissions engineering around a thin model core.

Paging is PagerDuty, through the Events API v2, with severity mapped and the dedup key carried through. Verified-customer incidents trigger the on-call engineer through PagerDuty's event-ingestion endpoint, with the server-assigned severity mapped onto PagerDuty's levels and the deduplication key computed in file-incident carried through unchanged — so duplicate calls about one outage do not double-page. The page is a deterministic consequence of a successful, verified filing, not a thing the model decides to send.

---

10. Security & Governance

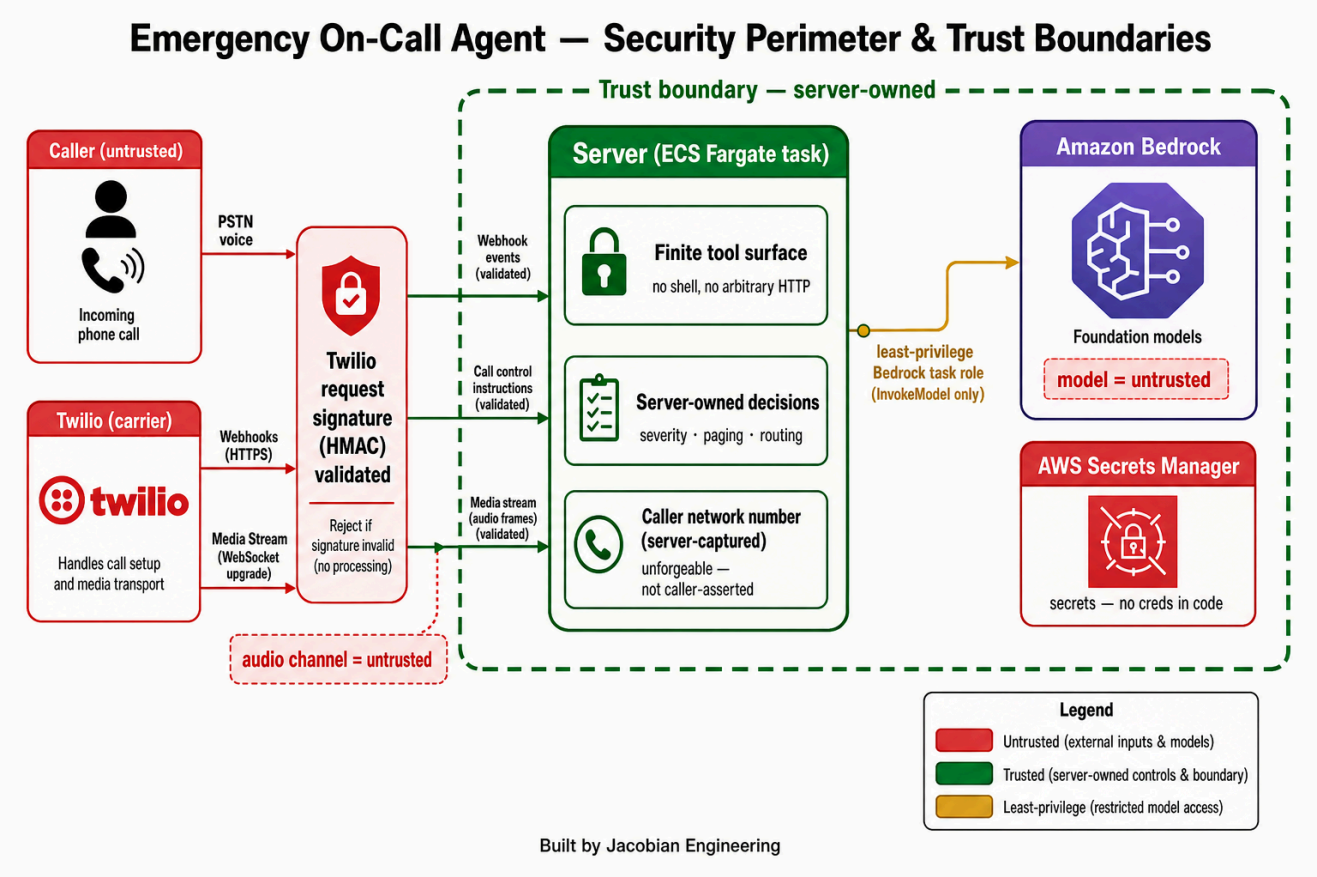


Figure 6 — The trust model: the audio channel and the model are both treated as untrusted, and every decision and identity-bearing field is server-owned.

The security posture rests on a single principle applied everywhere: the audio channel is untrusted, and the model is untrusted. Both assumptions are uncomfortable and both are correct. A phone number is reachable by anyone, and a generative model is, by nature, a thing that will produce plausible output under adversarial or merely confusing input. The architecture treats neither as a place to put trust, and the controls follow from that.

Inbound authentication is Twilio's request signature, and it is the perimeter. Every webhook and the media-stream upgrade are verified with Twilio's request signature — an HMAC keyed to a shared secret — so that only Twilio can trigger the service's side effects. This matters precisely because the carrier no longer publishes a stable IP range to allowlist; there is no network boundary to lean on, so the application-layer signature *is* the boundary. An unsigned or wrongly-signed request never reaches the logic that files tickets or pages engineers.

The action surface is finite, the decisions are server-owned, and the identity fields are unforgeable. The system can only do the five things in its tool catalog; there is no general-purpose, shell, or arbitrary-HTTP capability for the model to reach for, and the tool dispatcher enforces that only an agent's explicitly-enabled tools can be invoked. Whether to page, how to route, what severity to assign, and whether a filing succeeded are all computed in deterministic policy code, not at the model's discretion.

The caller's network number is server-captured and model-immutable. Cloud access follows least privilege: the task's Bedrock role carries only the model-invocation permissions it needs — with one documented resource-scope exception covered in §11.1 — and nothing more.

On governance, the agent embodies recognized AI-governance principles structurally rather than rhetorically. On **transparency and disclosure**, the agent identifies itself as an AI dispatcher in its greeting and discloses recording; callers are never deceived about interacting with a machine, which directly addresses the obligations frameworks such as the EU AI Act place on AI systems that interact with people. On **human oversight and escalation**, any caller who asks for a human, any unrecognized caller who warrants it, and any hard failure routes to a person; a fatal mid-call error escalates to a human line rather than dropping the caller. The system is designed never to strand a caller. On **bounded scope**, the agent is constrained to dispatch and nothing else — by architecture, not by prompt — so troubleshooting and advice are out of scope by construction. On **auditability and data governance**, every call yields structured logs of every spoken turn, every state transition, and every action and its result; every ticket carries a machine-authored internal record; the recording lives on the ticket and the carrier copy is deleted. That posture — map the risk, measure it through the logs and an out-of-scope counter, then manage it — is consistent with the NIST AI Risk Management Framework's map/measure/manage discipline.

> *"Filing the ticket, paging the engineer, and hanging up the phone are guarantees, not model choices. A caller who is told they have ticket number such-and-such has a ticket, because a deterministic code path created it — not because a model said the words. We learned that the hard way when the model delivered a perfect goodbye on a live call and then never actually hung up, and when it cheerfully told a caller it couldn't open a ticket it had every ability to open. You do not fix that with a better prompt. You fix it by making call-control deterministic and treating the model's confidence as untrusted input." — Erik D. Jones, Lead Architect & CEO, TrustEdge AI*

---

11. Engineering Challenges & Learnings

This section is the heart of the case study. Each item below cost real debugging on real calls, and each yielded a transferable lesson.

11.1 The cloud-only 403 that didn't reproduce locally. The single most expensive bug was an authorization failure that only appeared in production. The model's bidirectional-stream invocation returned access-denied when running under the production task role — despite an IAM policy that granted exactly that action on exactly the model's resource identifier, and despite the policy simulator reporting the action as *allowed*. On a developer's machine, with broad credentials, it worked perfectly. The

diagnosis: the Bedrock bidirectional-stream API authorizes against *more than the bare model ARN* — it internally references additional resources such as inference profiles — so a narrowly-scoped resource policy is insufficient even when the specific model identifier is correct. The fix was to broaden the *resource* scope for the model actions while keeping the *action* set minimal. Underneath that lived a compounding issue: the inherited code hardcoded a named local credential profile and only ever read credentials from environment variables, but production credentials arrive through the container platform's credential endpoint, not a named profile. A credential resolver that adapts the platform's standard credential chain to the model SDK's identity interface fixed it, so identical code now authenticates on a laptop and on the container's credential endpoint alike. The two lessons: streaming and composite APIs may authorize against resources beyond the obvious one, so **trust live behavior over the IAM policy simulator**; and credential acquisition is environment-specific, so build for the platform's credential chain from the start.

11.2 The model will not follow your script, so put the process in code. Covered in depth in §6, this was the defining architectural lesson. Two live calls proved that no amount of prompt forcefulness or temperature tuning would make a generative speech model reliably follow a multi-step intake script — it skipped verification and troubleshot regardless. The cure was not a better prompt; it was an application-level state machine that reduces the model to recognition and one-sentence-at-a-time delivery. Generative models are conversational surfaces, not process controllers.

11.3 The model will not reliably hang up, so guarantee it in code. On a live call the model delivered its scripted goodbye but never invoked the end-call action, leaving the line open for the better part of a minute until the caller hung up. After a couple of detour turns, it had simply compressed the closing sequence and dropped the action. The fix is a two-part backstop, armed only after an incident has actually been filed: a goodbye-phrase detector that fires when the agent's *own* speech contains a closing phrase — so a caller saying "take care" cannot trigger it — and an idle watchdog that hangs up if no model output has occurred for a set interval. Both deduplicate against each other and against the normal end-call path. Terminal call-control is a guarantee the business needs; it never depends on the model choosing to act.

11.4 The model claimed it couldn't do its job. Early calls produced a striking failure: the agent told callers it was *unable* to open a ticket or page anyone — a hallucinated limitation, since those are exactly its capabilities. The fix was explicit negative-capability framing in the prompt ("you can and do open tickets and page the on-call engineer; never tell the caller you cannot"). This is defense in depth — the real guarantee that filing happens lives in the deterministic file-incident path — but it stopped the model from verbally undermining a service it was, in fact, performing. Related prompt-compliance fixes followed the same pattern: determine severity silently and never speak it as a label, isolate the contact-capture step as its own mandatory turn because the model kept bundling and skipping it, and never read field labels or punctuation aloud. Each was a small wording fix, and collectively they underline that

prompt engineering is necessary but never sufficient — it shapes delivery, while code enforces guarantees.

11.5 State-dependent side effects must wait for the right lifecycle stage. The recording-start race from §7 — rejected because recording was attempted before the call was in progress — generalizes to a rule the whole codebase leaned on: issue a side effect at the lifecycle stage where its precondition actually holds, not at the earliest handler that is convenient. Moving recording to the media-stream handler resolved it cleanly, and the same discipline governs when the hang-up backstop is armed and when the call-to-ticket mapping is registered.

11.6 Inherited-codebase archaeology. The project began from a non-working prototype whose commit message was, literally, "still broken." A structured review found seven independent root causes: a one-way media fork instead of a bidirectional stream, so the model's audio never played; a hardcoded credential profile incompatible with the cloud runtime; a half-built barge-in with no queue flush; tools declared but never wired; audio-ordering hazards from per-chunk tasks; a removed standard-library audio module; and missing production basics — no signature validation, no health check, no logging, no tests. The lesson is mundane and important: a demo that "talks" is far from a service that "works," and the gap is ordering, authentication, lifecycle correctness, observability, and tests.

11.7 Latency, hidden. Several choices keep perceived latency low, and none of them is glamorous: end-to-end 8 kHz audio so no resampling is needed on the inbound path; running Nova Pro extraction concurrently with the next turn so its cost is invisible; masking multi-second ticket and paging API calls behind a short "one moment" audio clip; and returning from the end-call action immediately so the goodbye plays before the line drops. Together they are the difference between an agent that feels responsive and one that feels like software.

---

12. Complete AWS / Service Inventory

The complete service inventory for Jaiven, ordered from the voice and AI plane through the supporting telephony, compute, delivery, security, and observability fabric.

Category	Service	Role
AI/ML — voice plane	Amazon Bedrock	Single governed plane for every foundation-model call — speech conversation and text extraction alike.
	Amazon Nova 2 Sonic	Speech-to-speech model carrying the live spoken conversation over a Bedrock bidirectional stream (`amazon.nova-2-sonic-v1:0`).
	Amazon Nova Pro	Text model performing concurrent structured extraction of incident slots from the running transcript.

Category	Service	Role
Speech synthesis	Amazon Polly	"Matthew" neural voice, used offline to pre-render the deterministic greeting asset (not a runtime dependency).
Telephony	Twilio	Carrier for the emergency line — inbound webhooks, bidirectional Media Streams (mu-law), and REST control for transfer, hang-up, and recording.
Compute	Amazon ECS Fargate	Single long-lived containerized task hosting the entire service.
Container registry	Amazon ECR	Holds the service's container image.
Load balancing	AWS Application Load Balancer	Terminates TLS for HTTPS webhooks and the secure WebSocket media stream.
TLS certificate	AWS Certificate Manager	DNS-validated certificate for the service hostname.
DNS	Amazon Route 53	Hosted zone resolving the stable service hostname to the load balancer.
Secrets	AWS Secrets Manager	Runtime-injected credentials for ticketing, paging, and the carrier; no secret in image or source.
Observability	Amazon CloudWatch Logs	Structured logs of every spoken turn, state transition, action, and result.
IaC	AWS CDK (Python)	Declarative definition of the entire stack.
CI/CD	GitHub Actions + AWS OIDC	Keyless, federated deployment pipeline using short-lived credentials.
Ticketing	Jira Service Management Cloud	Per-customer Service Desk requests, priority/org tags, internal record comment, recording attached as a file.
Paging	PagerDuty (Events API v2)	On-call engineer paging with server-assigned severity and dedup key carried through.
Application framework	FastAPI + uvicorn	The service runtime and ASGI server.

There is no workflow-orchestration middleware in this inventory and no second model provider: the service calls Bedrock, Twilio, JSM, and PagerDuty directly, which keeps the AI plane governable as a single surface.

---

13. Outcomes & What the Design Demonstrates

Jaiven runs in production on TrustEdge AI's own after-hours emergency line as the division's flagship reference implementation. It answers the call, conducts a

natural conversation, files correctly-routed and deduplicated tickets, pages the on-call engineer, attaches the call recording to the ticket, and closes the call reliably — with no human in the dispatch loop and a full audit trail behind every action. Because TrustEdge built it for itself first, the failures that only surface under real telephone traffic were found and engineered out on TrustEdge's own number, not a customer's.

More than a feature list, the build demonstrates a methodology for productionizing voice AI that TrustEdge applies across its work. Let the model do one thing well — natural conversation and recognition, not state, decisions, actions, or guarantees. Split voice from data, so a second text model handles structured extraction and neither model does the other's job. Own the process in code, with a finite state machine that steers the model one sentence per turn, making the conversation deterministic without making it robotic. Make every guarantee deterministic — filing, paging, hang-up, escalation, and routing are code, not model choices, with backstops for the things the model is supposed to do but might not. Treat the channel and the model as untrusted: authenticate inbound, keep the action surface finite, and make identity-bearing fields server-owned and unforgeable. And engineer for the platform, observe everything, and trust live behavior over simulators.

For organizations in regulated sectors evaluating agentic AI, the takeaway is uncomfortable and clarifying: the model is the easy part. The hard, value-creating work is not the conversation — it is the deterministic engineering, the integration and permissions discipline, and the governance controls built *around* the model. A voice agent that impresses in a demo is a model talking; a voice agent that can be trusted on an emergency line at 2 a.m. is deterministic code that happens to talk through a model. That is the distinction this build was designed to make concrete.

Jaiven is also one half of a pair. TrustEdge's companion front-of-house **receptionist agent, "Chris,"** escalates support and emergency calls to this agent — routing a live caller to Jaiven when a call comes in after hours or when no human can be reached during them. From Jaiven's perspective, such a caller is indistinguishable from a direct caller; the handoff is simply another inbound call. The receptionist's side of that escalation is documented in its own companion case study, and the two together make a useful side-by-side on the speech-model-versus-cascade trade-off and on how far the "decisions belong in code" discipline carries across different voice workloads.

> *"We built Jaiven on our own emergency line as a research implementation, and that is the whole point — we are not selling a thing we have only ever demoed. We dogfooded it, broke it on real calls, and rebuilt it until the only thing the model does is talk. What we are productizing is not the talking model. It is the deterministic engineering and the governance around it, because that is the part that is actually hard, and that is the part that makes it safe to put in front of a person whose system is down." — Erik D. Jones, Lead Architect & CEO, TrustEdge AI*