Deepgram

EBOOK

The Definitive Guide to Voice AI Agents

Table of Contents

Table of Contents

CHAPTER 0

Introduction

Setting the Stage

Why Voice Agents Now

Voice is no longer a novelty in human–computer interaction. It has become a foundational interface. This shift is driven by advances in real-time AI, growing production adoption, and the inherent complexity of spoken interaction.

Recent breakthroughs across the stack have made high-quality voice agents viable at scale. Large language models now support low-latency, multi-turn reasoning. Synthetic speech has become natural and expressive. Most importantly, new approaches to conversational speech recognition have solved one of voice UX's longest-standing challenges: accurately determining when a speaker has finished talking. This shift, driven by infrastructure-level research, has fundamentally changed how voice agents manage turn-taking, timing, and responsiveness.

At the same time, market adoption has accelerated. Organizations with high volumes of inbound and outbound conversations are deploying voice agents to extend availability, reduce staffing pressure, and improve responsiveness. Falling costs for real-time AI models, combined with more composable infrastructure, have lowered the barrier to entry. What once required significant in-house engineering effort can now be built with modern APIs and specialized platforms.

The result is a clear inflection point: the technology is mature, demand is real, and the systems involved are complex enough to require serious engineering. Voice agents are now production-grade, real-time software systems.

About This Guide

This guide is Deepgram's definitive, opinionated playbook for designing, deploying, and operating real-time voice agents. We wrote this as practitioners, for practitioners. It reflects our perspective on how voice agents should be architected and run in production.

While many resources explain how to call a speech API or connect to a telephony provider, far less guidance exists on how to design voice agents as end-to-end systems. Voice agents are distributed, low-latency systems that must coordinate perception, reasoning, timing, and response under real-world constraints. This guide focuses on that system-level view.

This guide is for developers and engineers building voice agents, architects evaluating design and deployment trade-offs, and technical product leaders responsible for reliability, scalability, and user experience. If you are implementing or evaluating a voice agent in a real production environment, this guide is for you.

Scope and Structure

The guide is organized into modular sections that can be read linearly or used as a reference:

- **Foundations** explains how voice agents work, their core components, and why real-time voice systems are uniquely challenging.
- **System Design and Architecture** covers agent behavior, real-time UX, reasoning logic, telephony integration, and multilingual considerations.
- **Operational Excellence and Governance** focuses on reliability, testing, metrics, monitoring, compliance, and guardrails.
- **Applied Architectures and Patterns** presents reference architectures and real-world integration patterns.
- **The Future of Voice AI** explores emerging directions, including Deepgram's Neuroplex research on speech-to-speech systems.
- **Getting Started** outlines practical next steps and resources for different personas.
- **Appendices** provide reference material such as glossaries, event flows, and troubleshooting guidance.

Throughout the guide, we emphasize how the pieces fit together at a system level. You will not find exhaustive API syntax here. Product documentation already serves that purpose. Instead, this guide helps you build a clear mental model of real-time voice agents: how they work, how to design them well, how to operate them in production, and where the technology is headed.

With that foundation in place, we begin by examining how voice agents are constructed and how they behave in practice.

CHAPTER 1

Foundations

How Voice Agents Work

Anatomy of a Voice Agent

Understanding voice agents begins with examining their fundamental structure and behavior. This section breaks down what voice agents are, how they operate, and why building them presents unique challenges.

What Is a Voice Agent?

A voice agent is an autonomous, real-time conversational system that listens, reasons, and responds using natural spoken language. Users interact with it through speech, typically over a phone call, device microphone, or embedded application, and receive spoken responses in return.

Voice agents differ from traditional IVR systems, which rely on prerecorded prompts and rigid menus. They also differ from text-based chatbots, which operate without real-time audio constraints. Unlike those systems, voice agents must handle continuous audio streams, ambiguous turn boundaries, and strict latency requirements while maintaining a natural conversational flow.

A voice agent is a real-time system designed for spoken interaction under real-world conditions.

The Conversational Loop

At a behavioral level, every voice agent follows a continuous conversational loop that mirrors human dialogue:

Listen → Understand → Reason → Respond → Speak

These stages map to distinct system functions:

- **Listen:** Capture incoming audio from the user in real time.
- **Understand:** Extract meaning and detect conversational signals such as pauses and end-of-turn.
- **Reason:** Interpret intent and decide on a response or action.
- **Respond:** Generate spoken output.
- **Speak:** Stream audio back to the user and resume listening.

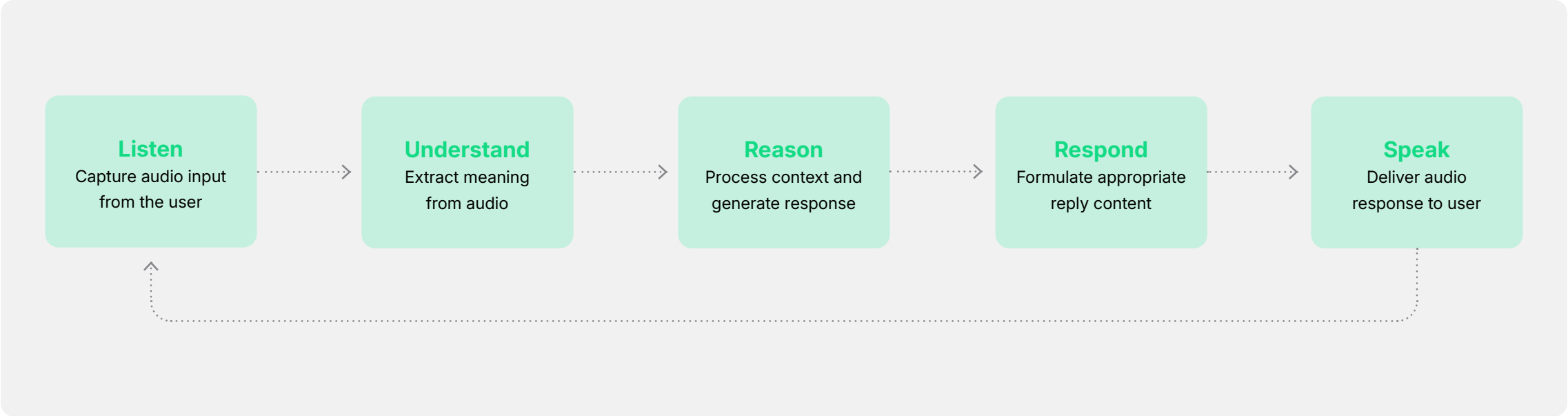


Figure 1.1: The Conversational Loop Flow
The five core stages of voice agent interaction form a continuous cycle, with each stage feeding into the next.

In production systems, this loop does not operate as a strict, turn-by-turn pipeline. The stages overlap and run concurrently. An agent may begin reasoning before a user has finished speaking, or start speaking while the remainder of a response is still being generated. This **streaming, event-driven behavior** is what gives voice interactions a natural rhythm.

This loop describes how a voice agent behaves. Implementing it reliably under real-world conditions is where complexity arises.

Why This Is Hard

Building a voice agent that feels truly conversational is difficult because real-world interaction is not a controlled environment. People interrupt each other, audio quality varies, latency introduces awkward pauses, and individual AI components do not inherently share timing or state. What appears simple quickly becomes a real-time systems problem.

The core challenges are:

- **Real-time orchestration of multiple systems**
  A voice agent consists of multiple independently operating components such as speech recognition, reasoning, synthesis, and audio I/O, each producing asynchronous events. Without strong orchestration, timing mismatches cause premature cutoffs, delayed responses, or the agent speaking at the wrong moment.
- **Latency compounding across stages**
  Small delays at each stage of the conversational loop accumulate into perceptible pauses. To improve responsiveness, overlap work, react to partial signals, and minimize handoff overhead. In voice interaction, every 100 milliseconds matters.
- **Interrupt-driven interaction and turn-taking**
  Users interrupt responses, correct themselves mid-utterance, and change intent while the system is speaking. A production agent must detect these events instantly, stop or adjust output, and resume listening without losing context.
- **Synchronizing speech and reasoning for natural rhythm**
  When reasoning takes time, the agent must manage perceptible states such as listening, thinking, and speaking. Without awareness of its own timing, systems produce awkward pauses, self-interruptions, or repetitive filler.

Together, these constraints make it insufficient to treat voice agents as sequential pipelines. Human-like interaction requires an **event-driven, interrupt-aware architecture** that treats **timing and partial signals as first-class inputs**.

The next section examines the systems that make this behavior possible, starting with the core technologies in the voice agent stack.

Voice Agent Stack

We can now examine what a voice agent is actually built from. While implementations vary, modern voice agents share a common set of architectural building blocks. These components exist to support the conversational loop described earlier and to operate reliably under real-time conditions.

The architecture described in this section represents cascade-based voice agents: systems that convert speech to text, process that text through a reasoning layer, then synthesize responses back to speech. While the industry is actively developing streaming speech-to-speech (S2S) architectures that operate on continuous audio representations without text intermediaries, cascade systems remain the dominant production pattern. They offer mature tooling, interpretable debugging boundaries, and proven operational characteristics.

For a detailed exploration of S2S architectures and Deepgram's Neuroplex research, see Chapter 7: The Future of Voice AI.

The Stack Concept

At a high level, a voice agent stack can be divided into two layers.

The **core conversational layer** contains the components required to carry out a real-time spoken interaction at all. This layer is responsible for listening, reasoning, and speaking, and for coordinating those functions continuously. Without it, there is no conversational agent.

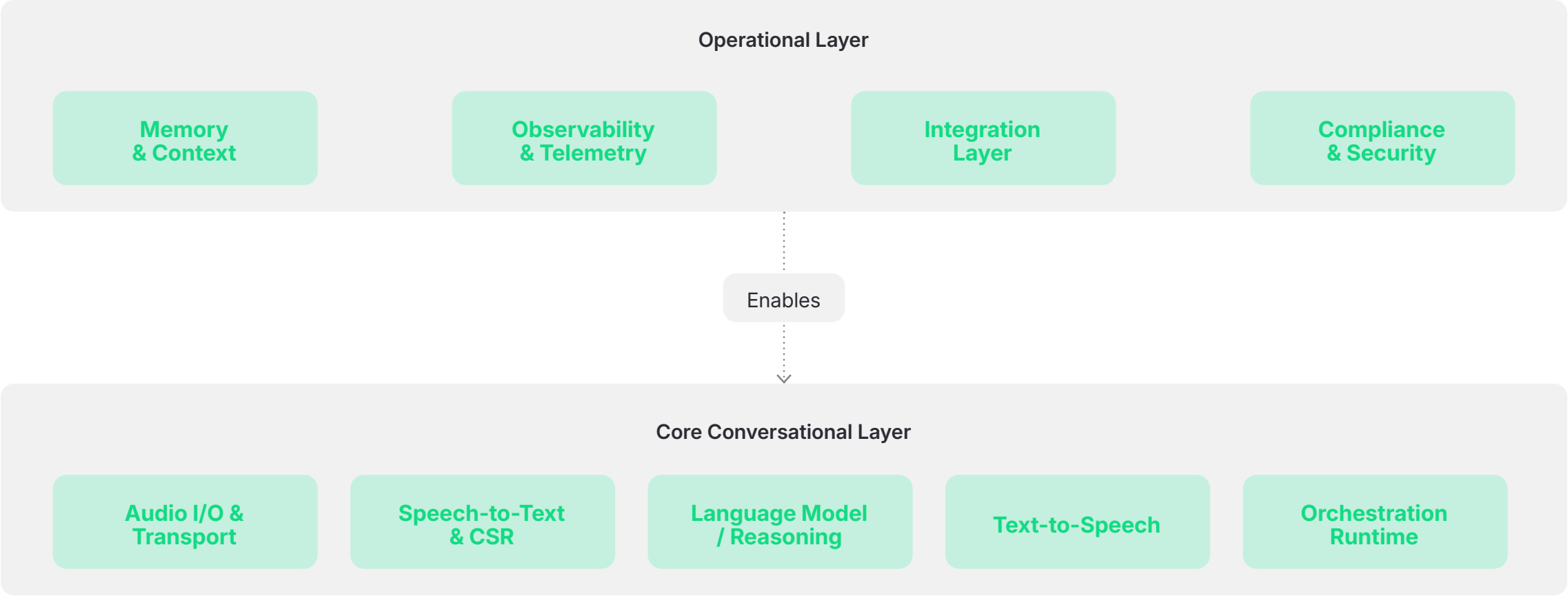
The **operational layer** sits above the core and becomes essential as systems move toward production. This layer handles concerns such as scalability, reliability, observability, integration with external systems, and compliance. These components do not change how the agent converses, but they determine whether it can function dependably in real-world deployments.

Put another way, the core layer answers the question: **Can the agent hold a conversation?**

The operational layer answers: **Can that conversation run at scale, under load, and within real-world constraints?**

The sections that follow break down these layers in more detail, starting with the minimal set of components required for a functioning voice agent, and then expanding to the additional capabilities needed for production-grade systems. Where relevant, we reference how Deepgram provides infrastructure across both layers.

Figure 1.2: Voice Agent Stack Architecture
The two-layer architecture separates production concerns (Operational Layer) from real-time interaction capabilities (Core Conversational Layer).



MVP Stack: Functional Core

At the foundation of every voice agent is a real-time conversational core. These components are non-negotiable. Without all of them working together, a system may demonstrate basic functionality but will not behave like a true conversational agent.

Component	Role in the System	Deepgram Example
Audio I/O and Transport	Streams audio into and out of the system with low latency.	Real-time, bidirectional audio streaming via **SDKs** and APIs for web, mobile, and telephony environments.
Speech-to-Text and Conversational Speech Recognition	Transcribes speech and detects conversational boundaries such as end-of-turn.	**Flux** unifies transcription and turn detection in a single conversational model optimized for real-time agents.
Language Model or Reasoning Engine	Interprets intent and determines the agent's response or action.	Model-agnostic LLM integration optimized for streaming input and incremental output.
Text-to-Speech	Converts responses into spoken audio and streams output to the user.	**Aura-2** provides low-latency, streaming speech synthesis for real-time interaction.
Orchestration Runtime	Coordinates timing, state, and interaction across all components.	**Voice Agent API** manages real-time orchestration, interruption handling, and stream control within a single session.

Together, these components form the minimum viable voice agent. Early implementations often stitched these pieces together manually using multiple APIs, which worked for demos but introduced unnecessary latency and coordination issues. Modern platforms increasingly integrate parts of this core to reduce handoffs and enable streaming, event-driven behavior by default.

Production-Ready Stack: Operational Layer

While the functional core enables conversation, production deployments introduce additional requirements. Voice agents must operate reliably at scale, integrate with business systems, and meet regulatory and security expectations. These concerns are addressed by the operational layer of the stack.

Production-Ready Operational Components

Component	Role	When Needed	Deepgram Example
Memory and Context Management	Maintains conversational state across turns or sessions.	Required for multi-turn or personalized agents.	Session-level context, designed to integrate with external stores you manage for long-term memory.
Observability and Telemetry	Measures latency, quality, and system health.	Required for any production deployment.	Real-time event streams that plug into your monitoring stack to support quality tracking and external observability.
Integration Layer	Executes actions in external systems.	Required for task-oriented agents.	**Structured function calls** that integrate external logic into the conversational flow.
Compliance and Security Controls	Enforces privacy, access control, and regulatory requirements	Required for enterprise and regulated environments.	**Regional deployment options**, isolated infrastructure, **short-lived credentials**, and redaction support.

These components do not change how an agent speaks or listens, but they determine whether the system can be trusted in real-world environments. Most successful deployments begin with the functional core and progressively add operational capabilities as they scale.

With the components of the voice agent stack defined, the remaining question is how these pieces should be assembled into real systems. There is no single correct approach. Different teams make different architectural trade-offs depending on their requirements, constraints, and appetite for control.

Deepgram's Opinionated View

Deepgram's perspective is that real-time voice agents perform best when built on an integrated, streaming-first speech infrastructure rather than a loosely chained set of APIs. Speech recognition, speech synthesis, and audio transport should operate as a unified layer, coordinated by a purpose-built orchestration runtime.

Optimizing the speech loop as a whole reduces latency, minimizes handoffs, and enables faster transitions between listening, reasoning, and responding. This produces more responsive and natural interactions than optimizing individual components in isolation.

At the same time, Deepgram does not advocate for a monolithic stack. Teams should remain free to choose their own language models, tools, and business logic. While reasoning layers may vary by use case, the speech layer must be unified and streaming-first to support interruption handling, precise timing, and low-latency response.

Deepgram supports different abstraction levels by offering flexible APIs that allow teams to decide how much of the system they want to manage themselves.

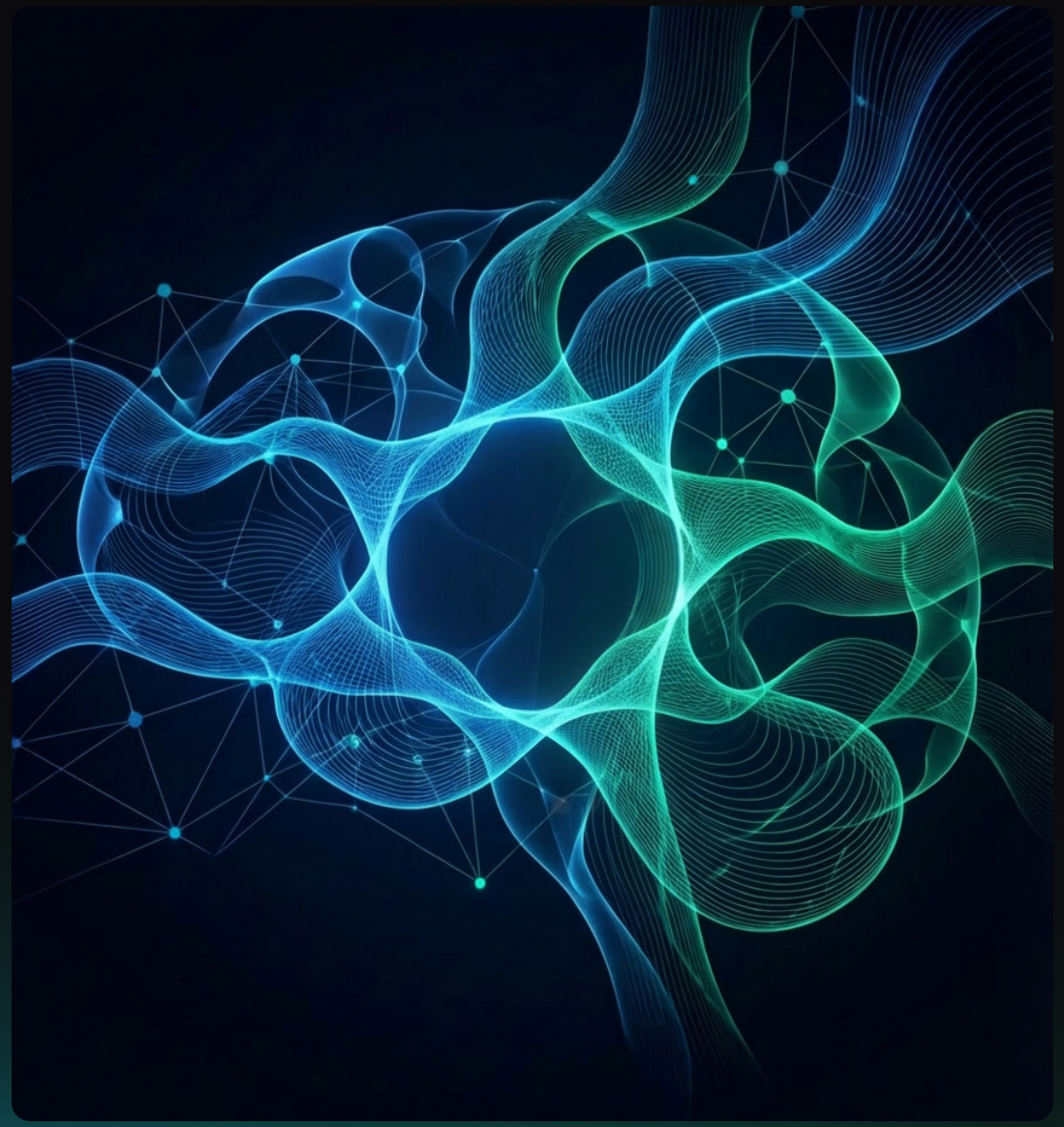
The next section examines how teams assemble these components in practice. There are multiple valid approaches, each with trade-offs in control, complexity, and abstraction.

Architectural Approaches to Building Voice Agents

While all voice agents rely on the same core components, teams differ in how they assemble those components. The primary differences come down to **where orchestration logic lives** and **how much abstraction versus control** a team prefers.

At one end of the spectrum, teams assemble everything themselves using low-level APIs. At the other, teams adopt fully managed platforms where voice is one feature in a broader system. Most real-world implementations fall somewhere in between.

Rather than rigid categories, these approaches represent **common architectural patterns**. Teams often move between them as requirements evolve.



Overview of Build Approaches

We can broadly group voice agent architectures into four patterns, ordered from maximum control to maximum convenience.

The Four Build Approaches

Build Approach	Description	Examples	Deepgram Role
DIY Frameworks (Custom Stack)	Teams assemble their own pipeline using separate STT, LLM, and TTS components and implement orchestration themselves or via open-source frameworks.	LiveKit, Pipecat, custom pipelines	Provides foundational STT and TTS APIs and SDKs used directly within custom orchestration code.
Unified APIs (End-to-End Runtime)	A single real-time API integrates speech, orchestration, and LLM interaction while still allowing configuration and custom logic.	Deepgram Voice Agent API, OpenAI Realtime API	Provides both speech infrastructure and real-time orchestration in a single runtime, with model choice and function calling.
Managed Voice Agent Platforms	Hosted platforms offering visual builders, telephony integration, and preconfigured orchestration for common use cases.	Vapi, Retell	Often used as the underlying speech layer powering platform-managed agents.
Enterprise Conversational Suites	Large CX platforms where voice is one channel within a broader, multimodal system.	Cognigy, Kore.ai, Decagon	Serves as the embedded speech infrastructure, typically consumed indirectly by end customers.

These approaches trade off control, abstraction, and operational responsibility. None is inherently better than the others. The right choice depends on team capability, use case complexity, and long-term ownership preferences.

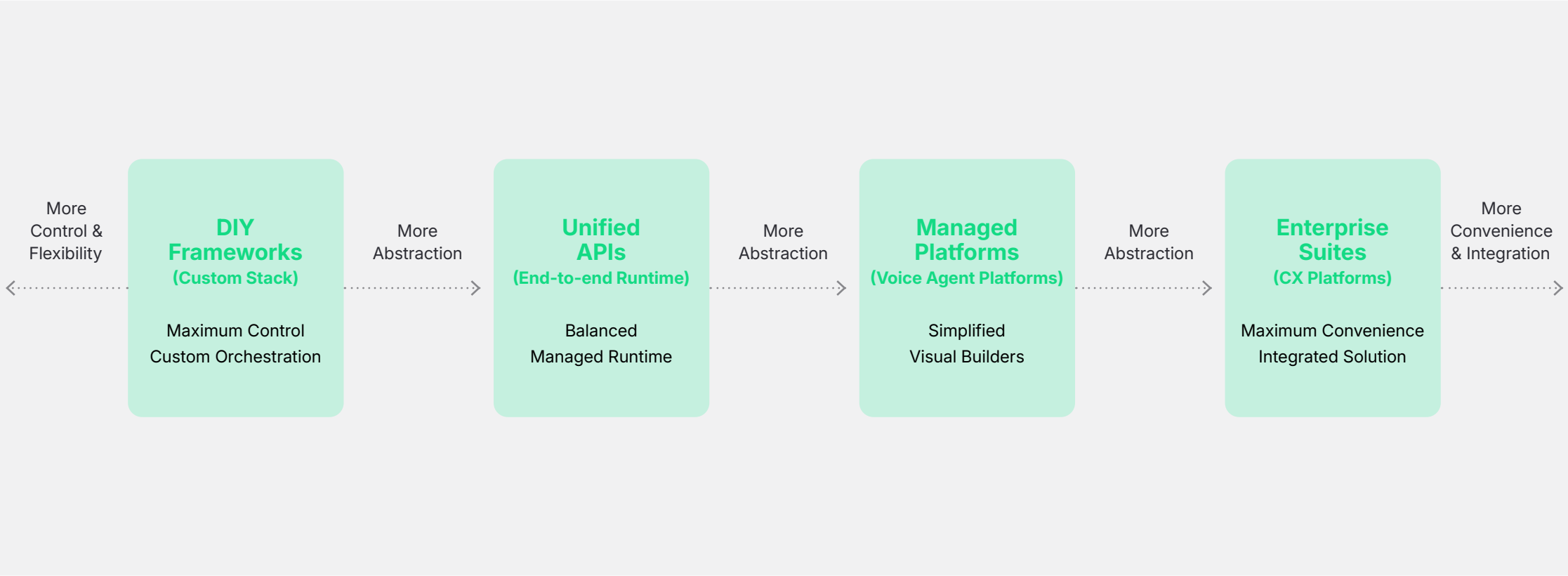


Figure 1.3: Build Approaches Comparison
The four architectural approaches form a spectrum from maximum control and flexibility (DIY Frameworks) to maximum convenience and integration (Enterprise Suites).

Comparing the Approaches

Rather than evaluating each approach in isolation, it is more useful to compare them along a few consistent dimensions:

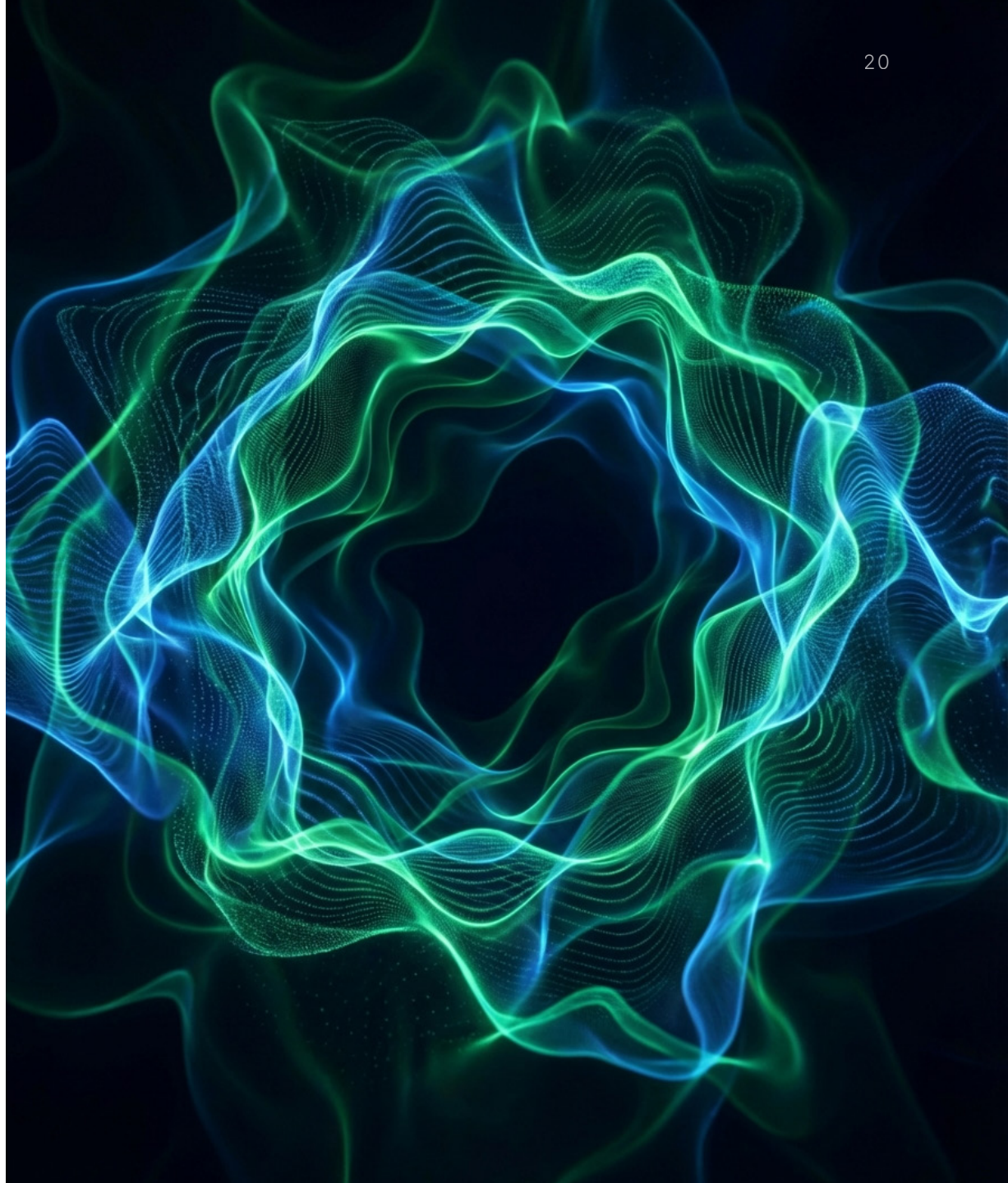
- **Abstraction level**
  DIY approaches offer the lowest abstraction and highest responsibility. Unified APIs strike a balance by abstracting orchestration while preserving code-level control. Managed platforms and enterprise suites offer higher abstraction, often at the cost of flexibility.
- **Customization and control**
  DIY provides full control over every component. Unified APIs allow meaningful customization within a managed runtime. Managed platforms limit customization to supported workflows. Enterprise suites offer the least direct control over AI behavior but maximize integration.
- **Latency and performance**
  In theory, DIY can be optimized for specific scenarios. In practice, purpose-built unified APIs often achieve better real-time performance due to integrated streaming and optimized internals. Higher-level platforms may introduce additional latency depending on telephony and routing layers.
- **Integration complexity**
  DIY places the full integration burden on the team. Unified APIs reduce complexity by collapsing the conversational loop into a single integration. Managed platforms simplify agent construction but may still require integration with downstream systems. Enterprise suites aim to minimize integration by providing a comprehensive solution.
- **Compliance and governance**
  DIY offers maximum flexibility but shifts compliance responsibility to the team. Unified APIs can provide strong governance options when deployed in controlled environments. Managed platforms are typically SaaS-first. Enterprise suites usually offer the most comprehensive compliance posture.

Guidance on Choosing an Approach

The right architectural approach depends on priorities and constraints:

- **Choose DIY** if you require maximum control, need to self-host components, or must integrate proprietary models. This approach demands strong engineering resources and careful real-time systems design.
- **Choose a unified API** if you want a balance of speed, flexibility, and performance. This is often the best fit for teams building production voice agents who want to avoid implementing streaming orchestration themselves while retaining control over models and logic.
- **Choose a managed platform or enterprise suite** if speed to deployment, minimal coding, or enterprise procurement requirements dominate. These approaches work well for standardized use cases or organizations that prefer integrated vendor solutions.

Many teams mix approaches over time. Some prototype with managed platforms and later transition to unified APIs or DIY stacks. Others embed unified APIs within larger platforms or enterprise workflows. Deepgram is designed to support this spectrum, allowing teams to adopt the abstraction level that best matches their needs.



CHAPTER 2

System Design and Architecture

Voice UX Design Principles

Voice user experience design sits at the intersection of linguistics, cognition, and real-time systems engineering. A voice agent does not merely transcribe speech and return answers. It must manage timing, tone, and conversational expectations so that interaction feels cooperative rather than transactional.

Across academic research and production deployments, one conclusion is consistent: **the quality of a voice agent is determined less by what it says than by *when* and *how* it says it**. Micro-timing, interruption handling, and perceptual cues shape whether users trust the system or abandon it.

Deepgram's approach to voice UX reflects this reality. We view voice interaction as a real-time control problem, not a text interface with audio attached. From that perspective, effective voice UX emerges from three interdependent design layers: **reactive**, **predictive**, and **adaptive** behavior. These layers govern how an agent responds in the moment, anticipates intent, and adjusts tone over time.

Reactive Design: Establishing Conversational Rhythm

Human conversation follows a surprisingly strict rhythm. Turn gaps that are too short feel interruptive; gaps that are too long feel inattentive. Reactive design focuses on reproducing this rhythm in software.

The foundation is **accurate turn-taking**. A voice agent must reliably determine when a user has finished speaking while tolerating natural pauses and hesitations. In production systems, this requires conversational speech recognition that models timing and intent, not just words.

Deepgram's conversational speech models are designed around this principle. Rather than treating end-of-speech as a fixed silence threshold, they infer turn completion from conversational context, allowing the system to respond quickly without cutting users off. This precision enables agents to maintain human-like pacing even under noisy or ambiguous conditions.

When downstream reasoning or data access introduces delay, reactive design calls for **explicit acknowledgment**. Short, low-commitment cues signal attentiveness without breaking conversational flow.

Conceptually, the behavior looks like this:

```
if user_turn_complete:
      prepare_to_respond()

if response_is_delayed:
      acknowledge_progress()

if user_resumes_speaking:
      immediately_yield_control()
```

The goal is reassurance through restraint. Silence creates uncertainty, while brief acknowledgment maintains trust.

Reactive design establishes the baseline rhythm of interaction. Without it, predictive and adaptive behaviors cannot compensate for broken pacing.

Predictive Design: Reducing Perceived Latency

Once basic rhythm is stable, voice agents can move beyond reaction to **anticipation**. Predictive design allows the system to infer intent before a user finishes speaking and begin preparing a response in parallel.

Research on incremental dialogue processing shows that overlapping listening and reasoning can dramatically reduce perceived latency. Deepgram's streaming-first architecture enables this behavior by emitting early conversational signals that allow downstream systems to act speculatively.

The core pattern is speculative preparation with safe cancellation:

```
if early_intent_inferred:
      begin_drafting_response()

if user_continues_or_changes_direction:
      discard_draft()

if turn_is_confirmed_complete:
      finalize_and_speak()
```

This overlap shortens response time without increasing error rates, provided the system can cancel cleanly when predictions are wrong.

Predictive design also shapes how speed is perceived. Users judge responsiveness not by absolute latency, but by feedback continuity. Subtle backchannel cues, brief affirmations, or tonal acknowledgment during processing can make identical systems feel significantly faster.

Deepgram's opinion is clear here: latency is as much a UX problem as an infrastructure problem. Optimizing models alone is insufficient if orchestration does not allow early signals to flow through the system.

Adaptive Design: Tone, Persona, and Context Awareness

Static voice personas break down quickly in real-world interaction. Effective agents adjust tone, pacing, and phrasing based on context while maintaining a consistent identity.

Adaptive design draws on two signal classes:

1. Input-side context, such as hesitation, interruptions, repeated questions, or unstable recognition
2. Output-side expressiveness, including prosody, pacing, and emphasis

Deepgram's speech models expose timing and stability signals that can be used to infer conversational friction. For example, rapidly changing partial transcripts or delayed stabilization often correlate with user uncertainty or frustration. These signals can guide downstream response shaping without explicit sentiment classification.

On the output side, modern expressive synthesis allows tone modulation without brittle markup. Deepgram's Aura-2 voices are designed to produce consistent, professional speech while adapting pacing and emphasis naturally from text and conversational context. Developers influence tone primarily through response construction and persona choice, rather than micromanaging synthesis parameters.

Vida's healthcare voice agents process tens of thousands of calls daily for appointment scheduling and medication adherence, where empathetic tone directly correlates with task completion. Their architecture prioritizes alphanumeric accuracy for dates and medication names without SSML markup, demonstrating that entity rendering precision is a UX requirement, not a feature.

The guiding principle is **contextual sensitivity, not emotional mimicry**. The agent should sound calm when users are rushed, reassuring when they hesitate, and concise when confidence is high.

Conversational Framing and Meta-Communication

Humans constantly talk **about** the conversation itself. These meta-communicative cues manage expectations and keep dialogue cooperative.

In voice UX, meta-communication reframes latency or uncertainty as progress rather than failure. Compare silence to a phrase like "I'm checking that now." The latter maintains engagement by explaining what the system is doing.

Meta-communication operates across layers:

Layer	Responsibility	Example
Policy	Decide when to acknowledge delay	"Acknowledge if reasoning exceeds one second"
Orchestration	Detect slow operations	Trigger acknowledgment
Timing	Ensure cues respect turn ownership	Never interrupt user speech
Synthesis	Convey reflective tone	Slight pause, measured delivery

The intent of meta-communication is transparency. When done well, it builds trust by making the system's internal state legible to the user.

Repair and Recovery

No voice agent avoids errors. Reliability depends on how gracefully the system recovers.

Effective repair follows a consistent pattern:

- Acknowledge uncertainty
- Narrow ambiguity
- Confirm resolution

Conceptually:

```
if confidence_is_low:
      ask_for_clarification()

if user_corrects_agent:
      prioritize_user_input()
      confirm_updated_understanding()
```

Interruptions during agent speech are especially important signals.
A system that immediately yields control reinforces that the user remains the primary speaker.

Deepgram's event-driven speech infrastructure is designed to support this behavior, allowing playback cancellation and rapid listening resumption without losing conversational state.

When recovery is transparent and cooperative, users perceive the agent as attentive rather than flawed.

Multimodal Reinforcement and Cognitive Load

Voice interaction often coexists with visual or tactile feedback. Coordinating these channels reduces cognitive load by reinforcing system state through multiple signals.

Auditory acknowledgments can be paired with visual indicators such as progress animations or status lights, all synchronized to conversational timing. The critical requirement is **temporal alignment**. When cues drift out of sync, trust erodes.

As multimodal interfaces mature, the best experiences will feel unified rather than layered, with users perceiving responsiveness rather than individual channels.

Perspective and Path Forward

Voice UX is shifting from scripted interaction toward real-time collaboration. Anticipatory processing, adaptive prosody, and event-driven orchestration are redefining how agents listen and respond.

Deepgram's position is that voice UX quality emerges from optimizing the entire conversational loop, not isolated components. Low-latency speech models, expressive synthesis, and fine-grained timing signals enable agents that feel cooperative rather than mechanical.

As these systems evolve, tone and timing will adapt fluidly to live context rather than fixed personas. Measuring and improving these behaviors in production is the next challenge, which we address in the following section.

Interaction and UX Architecture

If voice UX principles define **how an agent should feel**, interaction architecture defines **how that feeling is produced in real time**. This section focuses on the systems beneath the conversation: audio pipelines, event flow, interruption handling, and state signaling. A well-architected voice agent does not guess what is happening. It reacts deterministically to real-time signals and coordinates perception, reasoning, and output under latency and concurrency constraints.

Real-Time Audio Pipeline

Every voice interaction begins with audio capture, and architectural choices here directly affect responsiveness. Audio should be streamed in small, consistent frames rather than buffered in large chunks.

Frame sizes in the 20–50 millisecond range strike a balance between reactivity and network efficiency. Smaller frames increase responsiveness but raise overhead; larger frames reduce overhead but introduce audible lag.

Encoding formats should match the environment. Telephony typically uses 8 kHz mono μ-law or A-law PCM, while browsers and mobile devices commonly support 16 or 48 kHz PCM. On the output side, synthesized speech should be streamed incrementally and buffered just enough to prevent playback gaps. A short jitter buffer around 100 milliseconds is usually sufficient to smooth minor network variation without adding perceptible delay.

Deepgram's SDKs handle audio chunking, format negotiation, and stream continuity automatically, allowing developers to focus on interaction logic rather than transport mechanics. This abstraction matters because timing errors at the audio layer propagate upward into turn-taking and UX failures

Event-Driven Turn Management

Turn-taking is the backbone of conversational UX. Humans rely on implicit timing cues; systems require explicit events. A production voice agent should never poll for state. It should react to a stream of conversational events.

Key transitions such as speech start, speech end, interruption, and resume must be handled as asynchronous signals. These events drive both backend behavior and user-facing feedback. For example, detecting speech start should immediately cancel any active playback, while detecting turn completion should trigger reasoning or acknowledgment.

Flux simplifies this layer by collapsing traditional ASR and VAD stacks into a single conversational model that emits deterministic turn events:

- **StartOfTurn**: user begins speaking, cancel playback
- **EagerEndOfTurn**: medium confidence turn ending, begin speculative reasoning
- **TurnResumed**: user continues, cancel speculative work
- **EndOfTurn**: high confidence completion, finalize response

Thresholds such as eager_eot_threshold and eot_threshold allow teams to tune the speed–stability trade-off, and Flux is designed to keep transcripts highly consistent between eager and final turn boundaries. Speculative reasoning rarely diverges from the final text.

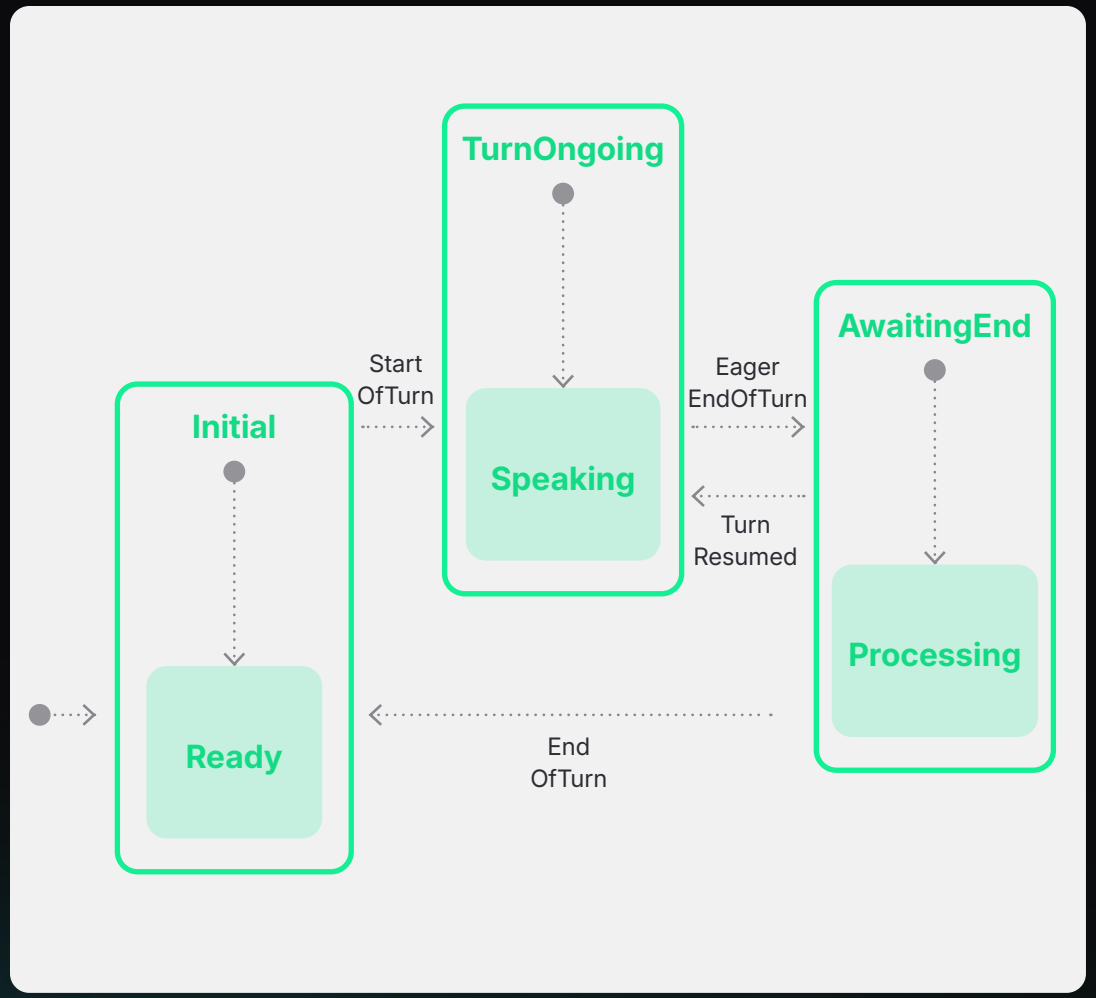


Figure 2.1: Flux Turn Management State Machine
Flux emits deterministic turn events, enabling the orchestration layer to react to speech boundaries without polling or redundant VAD.

When integrating Flux into frameworks like LiveKit, Pipecat, or Vapi, downstream VAD and turn logic should be disabled. Redundant detection introduces desynchronization, leading to premature responses or mid-utterance replies. Flux should be the single source of truth for conversational boundaries.

Deepgram's Voice Agent API builds on these primitives by exposing higher-level lifecycle events such as UserStartedSpeaking, AgentThinking, AgentStartedSpeaking, and AgentAudioDone. The underlying turn logic remains Flux-driven, but orchestration complexity is abstracted away, allowing teams to work at the interaction level rather than the signal-processing level.

Interruption and Playback Control

Interruption handling, or barge-in, must be supported at both the audio and orchestration layers. Users expect to interrupt naturally, and systems that fail here feel rigid and uncooperative.

Input and output pipelines must run concurrently. When new speech is detected during playback, output should stop immediately and state should transition back to listening. Playback cancellation must be explicit so buffers and downstream components reset cleanly.

In Deepgram's Voice Agent API, barge-in is handled automatically. User speech immediately cancels synthesis and playback, and the system re-enters a listening state. For custom implementations, inbound audio energy or speech detection flags should directly trigger playback termination. Confirmation events such as AgentAudioDone provide a clean boundary for recovery and next actions.

The core principle is control. TTS must be interruptible, cancellable, and replaceable at any moment. Voice agents that cannot stop themselves on demand will never feel conversational.

State Signaling and UX Feedback

Conversational UX depends on shared understanding of system state. Rather than inferring state from timing heuristics, production systems should emit explicit lifecycle signals.

Deepgram's Voice Agent API exposes structured events representing conversation state directly. These signals allow interfaces, analytics, and monitoring systems to remain synchronized without duplicating logic.

Treating state signaling as part of orchestration rather than presentation produces a single source of truth across devices, simplifying debugging and ensuring feedback aligns with actual system behavior.

Multi-Modal and Multi-Device Adaptation

Voice agents increasingly operate across browsers, mobile apps, contact-center consoles, kiosks, and embedded devices. Conversation logic should remain platform-agnostic while presentation adapts locally.

The orchestration layer should emit structured state and content, while clients decide how to render it. A display-equipped device may visualize

AgentThinking, while a voice-only interface inserts a brief acknowledgment triggered by the same event.

This separation allows consistent behavior across environments even as interfaces evolve.

Reliability, Failover, and Session Recovery

Voice interactions are long-lived and streaming-based, making transient failures unavoidable. Reliability architecture ensures that conversations degrade gracefully rather than collapse.

Streaming connections should implement reconnection logic with exponential backoff. When reconnection is possible, prior conversational context should be restored through injected state rather than restarting cold. When recovery fails, the system should communicate clearly rather than leaving silence.

Backend components should monitor audio flow and message cadence. If input or output stalls, the system should treat it as a failure and initiate recovery. Where possible, fallback responses or alternate synthesis paths should preserve conversational continuity.

The goal is not perfect uptime but resilient interaction. Even under failure, the agent should remain state-aware, transparent, and responsive.

Synthesis: Building Resilient Interaction

Interaction architecture is where voice UX becomes real. A production agent treats speech boundaries, interruptions, and reasoning delays as first-class events rather than edge cases.

By combining event-driven architecture with the UX principles described earlier, teams can build voice agents that remain fluid under real-world conditions. The result is steadier rhythm, better recovery, and interactions that feel attentive because the system never loses track of the conversation.

Reasoning and Orchestration Layer

Real-time voice agents succeed or fail in the layer that sits between perception and speech: the reasoning and orchestration layer. This is where transcripts become decisions, decisions become actions, and actions return to the conversation as coherent, timely responses. Unlike text-based systems, this layer must operate under tight latency constraints, tolerate interruption, and maintain conversational continuity across asynchronous events.

In a production voice agent, reasoning coordinates thought, action, and response within a continuous conversational loop.

From Conversation to Decision

Effective voice agents deliberately separate **interpretation** from **execution**.

The model's role is to understand intent and decide **what should happen**. Deterministic systems handle **how it happens**. Business rules, validation, permissions, and side effects belong in code. Open-ended interpretation, ambiguity resolution, and language generation belong in the model.

Most modern architectures therefore follow a **Think → Act** pattern:

- **Think:** Interpret the user's intent and decide whether a response requires action.
- **Act:** Execute that action deterministically and return results to the conversation.

Deepgram's Voice Agent API formalizes this separation by combining streaming transcription, contextual history, and structured function calling inside a real-time orchestration loop. The result is a system where language models express intent, but execution remains controlled, auditable, and interruptible.

Function Calling as the Action Interface

Function calling is the primary mechanism that allows a voice agent to act in the real world. Rather than generating free-form text instructions, the model emits structured requests that the orchestrator executes against external systems such as databases, APIs, or device controls.

This design turns the voice agent into an interactive system rather than a conversational endpoint. The model decides **which** action to take and **with what parameters**. The orchestrator decides **when**, **whether**, and **how** that action is executed.

At a conceptual level, the flow looks like this:

```
# Pseudocode illustrating Think → Act orchestration

on_user_end_of_turn(transcript):
      decision = llm.think(
            input=transcript,
            context=conversation_state
      )

if decision.type == “function_call”:
      emit_state(“AgentThinking”)
            result = execute_function(
            name=decision.name,
            args=decision.arguments
      )
      update_context(decision, result)
      speak(
            llm.respond(
            context=conversation_state
            )
)

      else:
      speak(decision.text)
```

This pattern highlights several opinionated principles:

- **The model never executes side effects directly**
- Every action has a clear request and response boundary
- Results are reintegrated into conversational context before speech
- The system remains explainable and auditable

Interruption-Aware Reasoning

Voice agents must remain responsive even while actions are in flight. If a user interrupts mid-task, the system should immediately prioritize listening over completion.

Architecturally, this means treating user speech as a higher-priority signal than pending actions:

```
on_user_started_speaking():
      cancel_active_action()
      stop_tts_playback()
      emit_state("Listening")
```

This interruption-first design prevents stale responses, reinforces user control, and preserves conversational trust. Long-running or asynchronous actions should always be cancelable or safely ignored if they become irrelevant.

Managing Latency and Perceived Responsiveness

Tool invocation introduces unavoidable delay. Each function call requires at least two reasoning steps: one to request the action and one to incorporate the result. In voice interaction, unmanaged silence during this gap quickly degrades user trust.

The orchestration layer should therefore manage perception as actively as execution:

- If reasoning or execution exceeds a short threshold, inject a brief acknowledgment.
- Treat progress cues as part of conversational flow, not error handling.
- Resume normal speech immediately when results arrive.

Because function calls are discrete messages, they are not inherently streamable. Execution should occur atomically, while the conversational layer maintains responsiveness through short, well-timed cues.

Reliability Under Real-Time Conditions

Voice agents stress reasoning systems more than text benchmarks suggest. Long sessions, overlapping turns, noisy input, and repeated tool use increase prompt complexity and error risk over time.

Production systems should explicitly evaluate:

- Correctness of function selection and arguments
- Stability under interruption and cancellation
- Time to first spoken response after user turn completion
- Consistency across long, multi-turn conversations

Determinism matters more in voice than in chat. Lower decoding temperature for transactional flows. Track state explicitly in code rather than relying on probabilistic memory. Log every decision boundary to support debugging, replay, and audit.

Memory and Context Management

Conversational coherence depends on disciplined context management. Short-term memory should retain recent turns and relevant function results. As sessions grow, older content should be summarized or pruned to preserve context limits without losing intent. Function call history can be retained explicitly, allowing the agent to reference prior outcomes naturally.

Long-term memory requires external storage. Persist only what is necessary, such as preferences or identifiers, and re-inject selectively at session start. Voice agents should remember **what matters**, not everything that was said.

Retrieval and Grounded Reasoning

For knowledge-intensive domains, retrieval-augmented reasoning grounds responses in verified data. In voice systems, retrieval belongs in the orchestration layer, not embedded directly in model prompts.

When retrieval is needed:

- Fetch small, high-relevance snippets
- Keep retrieval latency low
- Cache frequent queries
- Inject results into the next reasoning step

Deepgram's Voice Agent API integrates cleanly with external retrieval systems through structured function calls, allowing agents to reason over live enterprise data without sacrificing real-time responsiveness.

Building a Coherent Reasoning Loop

The reasoning and orchestration layer defines how a voice agent thinks and acts in real time. It governs how intent becomes execution, how actions remain interruptible, and how conversational state stays coherent under latency and uncertainty. When this layer is well designed, the agent feels responsive, deliberate, and in control.

However, all of this reasoning ultimately operates within a delivery environment that imposes its own constraints. For many production deployments, that environment is telephony.

Phone networks introduce additional complexity: constrained audio formats, strict timing expectations, call setup and teardown semantics, regional routing, and reliability requirements that differ fundamentally from browser or device-based voice interactions. These constraints shape how audio is streamed, how interruptions are detected, and how orchestration must behave at the edges of the system.

In the next section, we examine telephony architecture for voice agents. We explore how real-time voice systems integrate with PSTN and SIP infrastructure, how call state and media streams interact with conversational loops, and what architectural patterns are required to deliver natural, low-latency voice agents over the phone at scale.

Telephony Runtime Architecture

Telephony remains one of the most demanding environments for real-time voice agents. Inbound customer support, outbound automation, and compliance-sensitive workflows still rely heavily on PSTN and SIP infrastructure. Unlike browser or device-based voice, telephony introduces strict constraints around audio format, latency, session control, and call lifecycle management.

Building a production-grade phone agent requires more than connecting speech models to a phone number. It requires a runtime that can bridge legacy telephony systems with modern, event-driven voice orchestration while preserving natural conversational flow.

Bridging PSTN to Real-Time Voice Systems

A typical telephony flow begins when a caller dials a number managed by a provider such as Twilio. The provider terminates the PSTN call and establishes a bidirectional media stream, commonly via WebSocket, between the live call and a voice-agent runtime.

From that point forward, the call behaves as a continuous streaming session:

The telephony provider acts as a media bridge, converting PSTN audio into packetized frames and routing synthesized speech back into the call. Each session is identified by a unique call or stream identifier, which must be preserved on all inbound and outbound media to avoid cross-call contamination.

Deepgram's Voice Agent API is designed to sit directly behind this gateway, receiving and emitting audio streams while coordinating transcription, reasoning, and synthesis as a single real-time loop.

Audio Format and Latency Constraints

Telephony audio operates at 8 kHz, mono, using μ-law or A-law PCM. This limited bandwidth places a hard ceiling on acoustic fidelity and slightly increases recognition difficulty compared to broadband sources.

The most important optimization is **format alignment**. Avoid transcoding wherever possible. Deepgram's speech models accept μ-law PCM natively, and its TTS can emit μ-law audio directly consumable by telephony gateways. Eliminating format conversion removes unnecessary latency and reduces failure modes.

Even with optimal alignment, PSTN interactions typically add 100–200 milliseconds of round-trip delay. The best mitigation strategy is **regional proximity**:

- Match telephony media regions with speech infrastructure regions
- Co-locate orchestration middleware with both
- Minimize inter-region hops between gateway, agent runtime, and LLM

Telephony UX lives or dies on perceived responsiveness. Every avoidable millisecond matters.

Asynchronous Streaming and Event Coordination

Phone-based agents must send and receive audio concurrently. Blocking pipelines introduce dead air, clipped speech, or missed interruptions.

A robust telephony runtime uses non-blocking, event-driven loops for media ingress and egress:

```
async def receive_media(ws, inbound_queue):
      async for frame in ws:
      await inbound_queue.put(frame)

async def send_media(ws, outbound_queue):
      while True:
            frame = await outbound_queue.get()
            await ws.send(frame)
```

Recognition, reasoning, and synthesis operate independently but are synchronized through events. Signals such as “user stopped speaking,” “agent started speaking,” and “audio playback completed” anchor conversational timing and allow orchestration logic to manage fillers, interruptions, and state transitions.

Deepgram’s Voice Agent API surfaces these lifecycle signals explicitly, allowing telephony runtimes to remain reactive rather than polling-based.

Barge-In, Interruption, and Playback Control

Telephony users interrupt frequently. A phone agent that cannot stop speaking immediately when the caller interjects feels broken.

Barge-in must be enforced at two levels:

- **Media:** Stop or mute outbound audio the moment inbound speech resumes
- **Logic:** Cancel or invalidate any pending reasoning or tool execution

In integrated runtimes, interruption is automatic. In custom telephony stacks, inbound audio energy or speech detection should immediately preempt playback and return the system to a listening state. Playback confirmation events are essential to ensure buffers are reset cleanly before the next response begins.

DTMF and Call Control

Although speech-first interaction is preferred, telephony environments still produce keypad input. DTMF tones should be detected explicitly and handled outside the speech pipeline so they do not pollute transcripts.

Use DTMF sparingly and deterministically. Numeric shortcuts such as “Press 0 for an operator” should route cleanly into call-control logic while preserving conversational continuity for speech interactions.

Call termination, transfer, and escalation must also be handled explicitly. When a call ends or is redirected, the associated media stream should be closed immediately. During transfers, inform the caller verbally, mute audio during the transition, and avoid overlapping speech that can cause clipping or echo.

Voximplant’s native connector eliminates custom media infrastructure requirements for production voice agents across PSTN, SIP, and WebRTC channels. This architecture validates that abstracting telephony complexity at the platform layer accelerates deployment without sacrificing real-time control.

Escalation and Multi-Party Scenarios

In enterprise settings, AI agents frequently hand off to humans. The simplest pattern is a clean transfer: stop the AI stream and redirect the call to a new endpoint.

More advanced scenarios keep the AI present as an assistant while a human joins. These require strict separation of inbound and outbound streams to prevent cross-talk. The AI should never inject speech into the caller’s channel unless explicitly authorized.

Agent-assist architectures benefit from the same real-time transcription and orchestration capabilities, but the conversational contract changes. The AI listens continuously and advises silently rather than speaking.

Scaling and Concurrency

Telephony workloads scale horizontally. Each call maps to an independent, long-lived streaming session.

Scaling challenges typically arise in:

- WebSocket concurrency limits
- Orchestrator throughput
- Downstream LLM rate limits

Use async runtimes and load balancers that support connection persistence. Implement graceful connection draining and backpressure during traffic spikes. For critical flows, define fallback behaviors when upstream services degrade so callers never experience unexplained silence.

Monitoring and Quality Measurement

Telephony UX is judged almost entirely by timing and clarity. Measure:

- End-of-turn to first audio response latency
- Barge-in success rate
- Frequency of clipped or interrupted responses
- Call abandonment during silence

Event-level instrumentation provides the most actionable insight. Tracking user stop, agent start, and audio completion events across large call volumes reveals where conversational rhythm breaks down and where optimization yields the highest return.

Bringing It All Together

Telephony architecture is where legacy voice infrastructure meets modern real-time AI systems. Success depends on disciplined media handling, explicit event coordination, and interruption-first design.

When engineered correctly, the underlying complexity disappears. The caller experiences not a stitched-together system, but a responsive, conversational agent that behaves naturally despite the constraints of PSTN. That illusion is the benchmark of a production-ready telephony voice agent.

Multilingual Strategies and Localization

As voice agents expand globally, multilingual support becomes a system-level concern rather than a feature add-on. Language choice affects every layer of the stack, including speech recognition, reasoning, synthesis, orchestration, and UX. Successful multilingual agents balance accuracy, latency, cultural alignment, and operational simplicity.

The core challenge is preserving conversational continuity and persona while adapting to linguistic and regional variation in real time.

Language Strategy and Conversational Architecture

Multilingual voice agents typically follow one of two **conversation-level strategies**:

- **Unified multilingual conversational streams**, where a single real-time stream supports multiple languages dynamically
- **Language-specialized conversational streams**, where the system converges on a dominant language and optimizes for it over time

Examples:

Unified multilingual conversational streams: These are powered by a multilingual model that can understand and generate replies in many languages in the same session. For example, a customer support bot that responds in Spanish, then seamlessly switches to French if the user changes languages mid-conversation.

Language-specialized conversational streams: Some helpdesk or customer support systems require users to choose their language upfront. Once selected (e.g., Japanese), the voice agent uses a Japanese-specialized model for all understanding and response generation in that session.

Unified multilingual streams prioritize continuity. They avoid disruptive mid-conversation transitions, simplify orchestration, and handle moderate code-switching naturally. Language-specialized streams can optimize accuracy or cost in long-running interactions, but introduce additional complexity around transition, state management, and UX consistency.

Deepgram's platform supports both approaches, allowing teams to choose based on product requirements rather than model constraints. In practice, real-time voice agents benefit most from minimizing disruption. Continuity usually matters more than marginal accuracy gains unless regulatory, domain-specific, or cost considerations dictate otherwise.

Klubi's AI voice agents handle 30,000+ monthly interactions in Brazilian Portuguese across noisy mobile environments, maintaining 90%+ end-to-end conversation handling. Their deployment validates that domain-specific terminology accuracy and environmental robustness matter more than raw WER in production voice-led sales workflows.

Language Awareness and Adaptive Routing

Language awareness must emerge early in the interaction, but it should remain **incremental rather than decisive**. In real-time voice systems, language detection functions as a probabilistic signal that informs orchestration, not a hard routing gate that resets conversational state.

Streaming multilingual recognition allows agents to adapt dynamically as language stabilizes, without requiring explicit selection or disruptive handoffs. Early signals can guide response phrasing, acknowledgment style, or escalation logic while preserving turn-taking and conversational rhythm.

In more complex deployments, language confidence may still influence downstream behavior, such as:

- Restricting inference to a known language set
- Gradually specializing speech or reasoning behavior
- Triggering human handoff or fallback workflows

The key requirement is that language-aware behavior enhances responsiveness without breaking timing, interruption handling, or persona continuity.

Voice and Persona Consistency Across Languages

A multilingual agent should feel like the same character in every language. Inconsistent tone, pacing, or expressiveness breaks trust faster than minor recognition errors.

Persona continuity is best achieved by:

- Selecting voices with similar tonal characteristics across languages
- Maintaining consistent pacing and turn-taking behavior
- Treating voice selection as a UX and brand decision, not a technical one

Modern speech synthesis systems increasingly support expressive, enterprise-grade voices across many languages. Where native voices are unavailable, composable orchestration allows teams to integrate external synthesis providers while preserving real-time interruption handling and session control.

Language switching mid-conversation should update voice and synthesis behavior dynamically without resetting context. The user should experience adaptation, not reconfiguration.

Prompt and Context Localization

Localization is not translation.

Prompts, personas, and system messages should be authored directly in the target language. Relying on real-time translation introduces tone drift, syntactic artifacts, and cultural misalignment that compound over long interactions.

Effective localization requires:

- Language-native persona definitions
- Regionally appropriate politeness and formality norms
- Localized acknowledgments, clarifications, and error handling

Knowledge grounding must also be localized. If enterprise data exists in one language and the user speaks another, retrieved content should be translated and adapted before synthesis to avoid disruptive code-mixing.

Localization is a UX responsibility. Agents feel fluent when they follow linguistic norms rather than simply translating vocabulary.

Testing, Quality, and Fallback Behavior

Multilingual quality cannot be evaluated by transcription accuracy alone. Native speakers should assess pronunciation, phrasing, pacing, and cultural appropriateness. Each language may require different tuning for response length, acknowledgment patterns, or synthesis speed.

When an unsupported language is encountered, agents should fail gracefully:

- A brief explanation in the detected language, when possible
- Optional escalation to a human operator
- Clear acknowledgment rather than silence or confusion

Graceful degradation preserves trust even when full support is unavailable.

Summary

Multilingual voice agents succeed when language support is designed into the architecture rather than layered on later. Effective systems prioritize conversational continuity, persona consistency, and localization over rigid language pipelines.

Deepgram's platform supports this approach through real-time multilingual speech recognition, expressive synthesis, and a composable orchestration layer that adapts across languages without sacrificing responsiveness.

CHAPTER 3

Deployment and Runtime

Deployment and Runtime Architecture

Designing a high-performing voice agent does not end with model quality or UX design. Once deployed, success depends on infrastructure: where the system runs, how it scales under load, and how it behaves when things go wrong. Real-time voice systems place stricter demands on infrastructure than typical web services because latency, interruptions, and downtime are experienced directly through conversation.

This chapter focuses on **performance, availability, and runtime behavior**. Regulatory, privacy, and governance considerations are addressed separately in the Compliance chapter.

Hosting Models and Regional Placement

Voice agents must run close to users, support different levels of operational control, and operate reliably across environments. Deepgram supports these needs through multiple deployment models, including a fully managed Cloud API, single-tenant Dedicated clusters, EU-hosted endpoints for regional data residency, and self-hosted deployments for private cloud, bare-metal, or restricted environments.

The Cloud API provides the fastest path to production, with automatic scaling and global availability managed by Deepgram. Dedicated deployments give enterprises isolation, custom configuration, and predictable performance. Our EU-hosted endpoint provides deterministic regional data residency, ensuring inference and audio processing occur within a defined geographic boundary. In environments with strict connectivity or security constraints, self-hosted speech models can run locally while orchestration logic remains under customer control.

Endpoint proximity is critical for conversational responsiveness. Routing users to the nearest available speech endpoint can reduce round-trip latency by hundreds of milliseconds. Telephony gateways, orchestrators, and speech infrastructure should be co-located in the same region whenever possible to minimize inter-service hops and preserve conversational timing.

Scaling and Concurrency

Each active voice session maintains long-lived streaming connections for audio input, transcription, and synthesis. While a single session consumes modest resources, large deployments must support thousands of concurrent conversations without degradation.

Deepgram's managed speech infrastructure automatically handles concurrency and scaling for transcription and synthesis workloads. Customer-managed orchestration and reasoning layers should be designed with similar elasticity, using asynchronous runtimes and stateless gateways to manage large numbers of open connections efficiently.

Production systems commonly distribute sessions across a pool of compute behind a load balancer, ensuring that no single node becomes a bottleneck. External reasoning components such as LLM APIs may impose rate limits or cost constraints, so many deployments tier model usage, reserving larger models for complex turns and using lighter models for routine exchanges.

Resilience and Graceful Degradation

Real-world reliability is defined not by the absence of failure, but by how the system behaves when components degrade. Voice agents should never fail silently.

If reasoning or retrieval fails, the orchestrator should acknowledge the issue verbally and recover or escalate. If synthesis fails, a fallback voice or canned audio response should be used. If transcription confidence drops, the agent should prompt for clarification rather than proceeding with uncertain input. When automated recovery fails, the system should escalate to a human operator rather than trapping users in conversational dead ends.

Infrastructure changes should also be non-disruptive. Deploy updates gradually, drain existing sessions before recycling instances, and allow active conversations to complete cleanly. These practices preserve trust even during maintenance or partial outages.

Observability and Runtime Visibility

Operational insight is essential for both debugging and optimization. Every session should emit structured events capturing transcript segments, timing, interruptions, and major system transitions. These signals allow teams to correlate infrastructure behavior directly with user experience.

For self-hosted deployments, including those running on AWS SageMaker, private cloud, or bare-metal infrastructure, Deepgram exposes detailed event-level telemetry for speech activity, which can be integrated directly into standard observability stacks such as Datadog or CloudWatch. Tracking metrics like end-to-end response latency, interruption success rates, and session stability enables teams to identify regressions and continuously refine conversational performance.

These same event streams later serve as inputs to audit and governance workflows discussed in the Compliance chapter.

Edge and Offline Deployments

Some environments cannot rely on persistent cloud connectivity. Vehicles, kiosks, and industrial systems may require local inference to guarantee availability.

Deepgram supports these scenarios through self-hosted speech deployments that run STT and TTS models locally while preserving the same streaming interfaces used in cloud environments. These systems often trade model breadth for predictability, using constrained vocabularies or lighter models to maintain responsiveness on limited hardware. While less flexible than cloud deployments, edge architectures ensure continuity when network access is unreliable or unavailable.

Security as a Runtime Baseline

All real-time voice systems must operate over secure channels. Encrypted transport, short-lived authentication tokens, and strict network isolation are baseline requirements.

Deepgram enforces these controls across its Cloud, Dedicated, and EU-hosted deployments, while customer-managed components must implement equivalent safeguards. Security establishes the foundation for safe operation. How these controls extend into retention, access governance, and auditability is addressed in the next chapter.

Summary

Deployment is where voice UX becomes operational reality. The runtime architecture must balance latency, scale, and resilience without sacrificing availability or responsiveness. With thoughtful regional placement, elastic scaling, and graceful recovery strategies, teams can deploy voice agents that remain trustworthy and performant in production, regardless of scale or environment.

CHAPTER 4

Operational Excellence

Reliability, Testing, and Evaluation

Reliability for voice agents is defined by whether the system behaves naturally under real-world conversational conditions, not by uptime alone. Voice agents fail in experiential ways: replies that start too early, pauses that stretch too long, or responses that technically answer but miss intent. Ensuring reliability requires validating not just correctness, but conversational behavior.

Effective testing combines two complementary dimensions. Quantitative evaluation measures timing, flow, and responsiveness at the event level. Qualitative evaluation assesses meaning, intent alignment, and task success. Together, these approaches predict how an agent will feel to users in production.

Deepgram's Voice Agent API exposes detailed event telemetry that enables precise timing analysis. For large-scale simulation and behavioral evaluation, teams often pair this telemetry with tools from Coval, which specializes in probabilistic testing and multi-turn agent evaluation.

From Deterministic QA to Probabilistic Reliability

Traditional QA assumes that identical inputs yield identical outputs. Voice agents violate this assumption by design. Speech recognition, language models, and real-time orchestration all introduce variability.

The goal of reliability testing is therefore not to validate a single correctresponse, but to measure outcome distributions and improve their consistency.

Probabilistic evaluation treats conversational behavior statistically. Teams run many sessions under controlled variation such as accents, noise, or speaking cadence and measure how often the agent behaves acceptably. This approach reflects real usage and allows systems to be tuned for conversational smoothness rather than brittle correctness.

VAQI: Measuring Conversational Flow

The Voice Agent Quality Index (VAQI) provides a concise way to quantify conversational responsiveness using three timing behaviors:

- **Interruptions (I)**: how often the agent speaks before the user finishes
- **Missed responses (M):** how often the agent fails to respond within a defined window after a turn ends
- **Latency (L):** elapsed time from UserStoppedSpeaking to AgentStartedSpeaking

These metrics are derived directly from event timestamps, making them well suited for automated testing. Improvements in VAQI correlate strongly with perceived experience. For example, teams often see measurable gains after tuning end-of-turn thresholds or optimizing orchestration latency.

Retell AI consistently achieves ~800ms response times with interruption handling in production voice agents, demonstrating that sub-second responsiveness is achievable at scale when perception, reasoning, and synthesis are tightly orchestrated. This validates VAQI latency metrics as measurable predictors of conversational quality.

In large-scale benchmarks conducted with Coval, Deepgram's conversational speech models demonstrated faster response onset while maintaining transcription accuracy under production-like conditions, reinforcing the importance of timing as a first-class reliability metric.

Designing Effective Evaluation Programs

A comprehensive reliability program blends multiple testing methods, each revealing different failure modes:

- Probabilistic regression: Run many sessions under varied conditions and compare metric distributions across versions to detect drift.
- Replay testing: Reprocess recorded audio through new builds to isolate timing regressions while holding input constant.
- Load and stress testing: Simulate concurrent sessions and track tail latency rather than averages, since worst-case delays dominate user perception.
- Fault injection: Introduce controlled failures such as delayed reasoning or dropped synthesis to confirm graceful recovery.
- Turn-level diagnostics: Visualize UserStoppedSpeaking to AgentStartedSpeaking intervals and overlay interruption events to pinpoint orchestration bottlenecks.

Well-run teams maintain a fixed library of representative test calls and enforce tolerance bands, such as maximum acceptable VAQI deltas or missed-response rates, as part of every release cycle.

Qualitative Evaluation and Semantic Accuracy

Timing metrics describe how an agent behaves, but not whether it says the right thing. Qualitative evaluation measures intent alignment, semantic correctness, and tone. Coval supports this layer through a mix of automated checks and human-in-the-loop review.

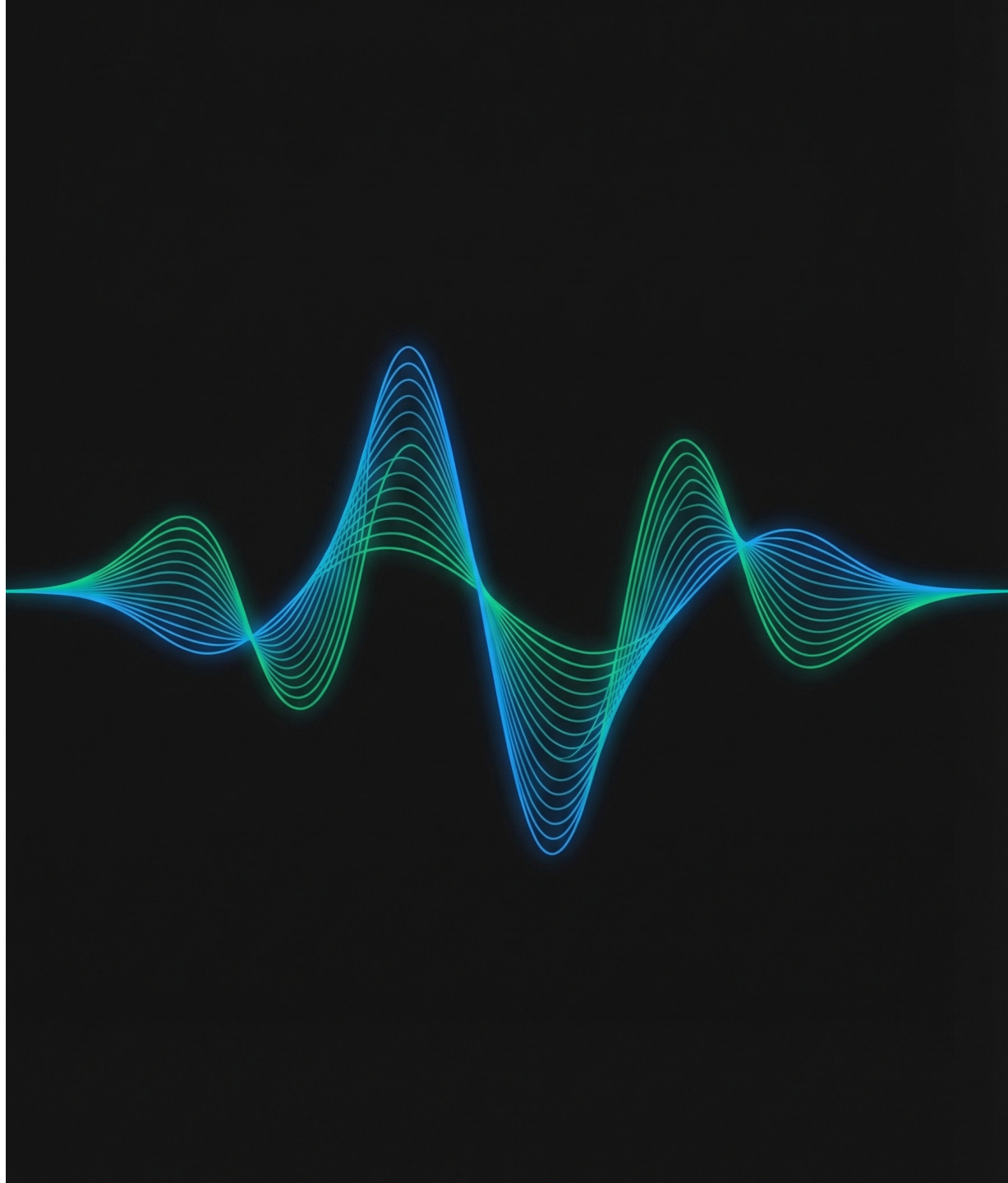
Common practices include scenario-based simulations, dialogue-level assertions such as required fields in responses, and human review of tone or empathy. These methods surface issues that quantitative metrics alone cannot capture, particularly in regulated or customer-facing domains.

Validating Reliability Before Production

Reliability must be proven before exposure to real users. Testing environments should mirror production configurations, including telephony codecs, streaming pipelines, and orchestration logic. New builds should be validated against baseline metrics and exercised under failure conditions such as transient network loss or delayed downstream services.

A release should advance only after key indicators such as VAQI, missed-response rate, and instruction compliance meet predefined thresholds. Once validated, these same metrics can be monitored continuously in production to detect drift and guide ongoing optimization.

Reliability begins with evaluation, but it is sustained through measurement. The next section extends these principles into production observability and continuous monitoring, ensuring that voice agents remain consistent as they scale.



Observability and Monitoring

Reliability testing validates a voice agent before release. Observability ensures that conversational health holds in production as traffic scales, models evolve, and real-world conditions vary. In real-time voice systems, latency, interruptions, and failures are immediately audible to users, which makes early detection essential.

Observability is concerned with detection and response. Its purpose is to surface live degradation quickly so teams can investigate, mitigate, or route traffic appropriately while conversations are in progress.

Effective observability turns conversational behavior into operational signals. It exposes emerging issues before users report them and provides the feedback loop required to maintain stable performance over time.

From Testing Metrics to Live Signals

Many of the same event boundaries used during testing continue to matter in production. These include turn boundaries, reasoning states, and playback transitions emitted by the Voice Agent API, such as UserStoppedSpeaking, AgentThinking, and AgentStartedSpeaking.

In production, these events function as live signals rather than evaluation artifacts. Tracking the intervals between them allows teams to monitor conversational timing and flow continuously. Aggregated across sessions, these measurements reveal shifts in responsiveness or turn handling that warrant investigation.

A production observability pipeline typically includes:

- Event instrumentation that emits timestamps and metadata for every user and agent turn
- Metric derivation that computes per-stage latency for speech recognition, reasoning, and synthesis, as well as end-to-end turn timing
- Storage and visualization using systems such as Datadog, Grafana, or CloudWatch
- Alerting based on sustained deviations, such as rising tail latency or interruption failures

When tuned correctly, this pipeline functions as an early warning system for conversational degradation.

Monitoring Conversational Behavior

Infrastructure health alone does not capture the quality of live voice interactions. Observability must also track how conversations unfold in practice so teams can detect abnormal patterns as they emerge.

Key behavioral signals include:

- Barge-in handling, measured by whether the agent stops speaking promptly when the user resumes
- Silence or stall detection, where expected responses fail to arrive within defined time windows
- Error distribution across speech recognition, reasoning, and synthesis components
- Session dynamics such as unusually long turns, repeated prompts, or looping behavior

These signals surface deviations from expected interaction patterns that may indicate upstream failures, orchestration regressions, or dependency issues. Correlating them with latency and error metrics helps teams localize problems quickly.

Dashboards, Alerts, and SLOs

Dashboards consolidate telemetry into a live operational view. Common views include active session volume, latency percentiles, interruption success rates, and error counts. Trend analysis often reveals issues before users experience widespread impact.

Service Level Objectives should express acceptable operational bounds for conversational systems, such as:

- Response latency remaining below a defined percentile threshold
- Interruption handling success staying within tolerance bands
- Error rates remaining below predefined limits

Alerts tied to these objectives exist to trigger operational response. They signal when investigation, mitigation, or traffic control actions are required, rather than serving as judgments of conversational correctness.

Logging and Traceability

Metrics reveal trends, while logs support incident investigation. Each conversation should carry a unique session identifier that links transcripts, events, and errors across systems.

Structured logging enables reconstruction of problematic interactions when diagnosing failures. Logs commonly capture system events such as retries, timeouts, and function calls, as well as conversational events such as speech boundaries and interruptions.

Transcript storage used for debugging must follow strict privacy controls. Sensitive data should be redacted, logs encrypted at rest, and access governed by role-based permissions, as described in the Compliance chapter.

Synthetic and Real-User Monitoring

Production observability benefits from combining synthetic probes with real-user telemetry.

Synthetic monitoring runs scripted sessions on a schedule to validate baseline system behavior. These probes often reuse representative scenarios developed during pre-release testing and help catch known failure modes early.

Real-user monitoring aggregates signals from live traffic. It captures variability introduced by user behavior, network conditions, and external dependencies that synthetic tests cannot fully simulate.

Together, these approaches provide both predictability and coverage. Synthetic monitoring confirms expected behavior, while real-user telemetry surfaces emergent issues.

Closing the Loop with Evaluation

Observability supplies real-world signals that inform what should be investigated, mitigated, or prioritized for further testing. It does not replace evaluation or testing programs. Instead, it provides continuous input that guides where deeper analysis is required.

By feeding production signals back into testing workflows, teams can refine scenarios, adjust thresholds, and focus evaluation efforts on the failure modes that matter most in practice.

Operational Discipline

Observability supports resilience only when paired with operational rigor. Teams should regularly validate alerting pipelines, rehearse incident response, and evolve dashboards as systems change. Deployments should be versioned and metrics tracked alongside build identifiers to preserve traceability.

Testing establishes readiness before launch. Observability maintains conversational health during live operation.

CHAPTER 5

Compliance and Governance

Compliance and Data Control

Operating voice agents responsibly requires protecting user data across its entire lifecycle, from capture and processing to storage and deletion. Compliance functions as an architectural property of the system and must be designed deliberately.

Many of the mechanisms referenced in this chapter, such as regional isolation, secure transport, tokenized access, and event logging, appear earlier as runtime capabilities. In this section, those same mechanisms are treated as governance controls that define how data is protected, retained, and audited over time.

Voice agents process sensitive data including live audio, transcripts, and user-specific information. Privacy and security therefore must be enforced deliberately at every layer of the stack. A compliant deployment is defined by how data flows through the system, how long it persists, and how access is controlled.

Regional Deployment and Data Residency

The execution environment of a voice agent determines which regulatory frameworks apply. Jurisdictions such as the EU and UK impose strict requirements on where audio and transcripts may be processed and stored.

Regional alignment ensures that audio and transcript data remains subject to the appropriate jurisdictional controls. Managed speech infrastructure should support explicit regional routing and avoid cross-region processing.

Vida processes hundreds of millions of TTS characters monthly for healthcare voice agents while maintaining HIPAA compliance, achieving 50% cost savings through unified STT+TTS from a single provider. This architecture demonstrates that compliance and cost optimization reinforce each other when vendor consolidation reduces both data exposure and integration complexity.

Highly regulated environments often require additional isolation. Single-tenant or region-specific deployments provide stronger data segregation, clearer audit boundaries, and more predictable governance. Deepgram supports this model through Dedicated clusters and EU-hosted deployments.

When external services such as language models or downstream APIs are involved, their residency guarantees must be evaluated as part of the overall compliance architecture. In environments requiring full control, self-hosting components within a controlled infrastructure provides the most deterministic guarantees.

The core principle remains consistent. Systems should be designed so data remains within the regions where it is legally or contractually required to reside.

Secure Transmission and Authentication

Protecting data in motion and enforcing strict access controls form the foundation of compliance. All real-time audio and transcript traffic should be encrypted using TLS, including secure WebSocket connections. Internal service-to-service communication should also use encrypted channels.

Authentication should rely on short-lived credentials rather than static API keys. Deepgram uses token-based authentication with ephemeral credentials that expire quickly, limiting exposure if intercepted. Long-running sessions should refresh tokens automatically without extending credential validity windows.

Client applications should avoid embedding long-lived credentials. Backend services should issue scoped, time-limited tokens following the principle of least privilege across browser, mobile, and device environments. The specific implementation of these controls varies by deployment model and is treated as an operational concern.

Data Retention, Redaction, and Minimization

Effective compliance places strong emphasis on data minimization.

Clear data-retention policies should limit storage to what is strictly necessary, with deletion or anonymization applied once data has served its purpose. Voice transcripts frequently contain personally identifiable information. Deepgram's speech-to-text APIs support model-based PII redaction, allowing sensitive entities to be masked at inference time.

Many organizations choose to avoid storing audio altogether. Deepgram does not retain audio by default, and storage is opt-in only for specific programs. Other deployments maintain short-lived buffers for quality assurance and delete recordings automatically. These minimization practices are essential for meeting GDPR, HIPAA, and PCI requirements.

When transcripts or logs are stored, they should be treated as confidential assets. Data should be encrypted at rest, protected by role-based access controls, restricted through IAM policies, and monitored through comprehensive access auditing. Storage systems should be private by default.

User Authentication, Consent, and Disclosure

When agents handle user-specific data or perform sensitive actions, users must be authenticated before proceeding. In telephony environments, this may involve caller ID verification, PINs, or step-up authentication flows. If authentication fails, the agent should restrict disclosures or escalate to a human operator.

Consent and disclosure requirements apply across many jurisdictions. Users may need to be informed that calls are recorded or that they are interacting with an AI system. Telephony frameworks can deliver standardized disclosures automatically, and agents should identify themselves as AI when asked.

Clear disclosure practices reduce regulatory risk and reinforce user trust.

Logging, Auditability, and Governance

Compliance requires verifiable records of sensitive system behavior. Audit logs should be maintained for actions such as financial transactions, consent capture, and content filtering. These logs should include timestamps, session identifiers, and relevant metadata, and should be designed to prevent tampering.

Audit logs typically reuse the same event streams captured for operational observability, with stricter retention, access controls, and immutability guarantees applied. This alignment ensures consistency between operational monitoring and compliance reporting.

Access to audit data should be tightly restricted, encrypted at rest, and stored using immutable or versioned storage where appropriate. Organizations should maintain documented incident-response and business-continuity plans and validate them regularly.

Security and compliance operate as continuous processes that evolve alongside the system.

Putting Compliance into Practice

Compliance for voice agents is achieved through layered controls:

- Keep data within the appropriate region
- Secure data in transit and at rest
- Minimize and redact stored information
- Authenticate users and capture consent
- Log sensitive actions for accountability

Deepgram's platform supports these principles through regional endpoints, tokenized authentication, built-in PII redaction, and deployment options such as Dedicated and EU-hosted runtimes. These capabilities align with common enterprise compliance frameworks including SOC 2 Type II, HIPAA, PCI, GDPR, and CCPA.

Compliance ultimately reflects a design philosophy. Systems built with privacy by default, disciplined data handling, and consistent governance controls can operate at enterprise scale while maintaining trust, safety, and regulatory alignment.

Content Safety and Guardrails

Voice agents operate in open-ended, real-time conversations, which makes them uniquely exposed to unsafe, sensitive, or out-of-scope inputs. Unlike traditional software, failures in voice systems are immediately perceptible and cannot be silently corrected. Once something is spoken, it cannot be taken back.

Content safety is therefore not an add-on. It is a core systems responsibility that protects users, organizations, and operators by enforcing clear behavioral boundaries at runtime.

Effective guardrails operate across three layers: input moderation, reasoning control, and output filtering. These technical controls are reinforced by governance processes that track, review, and continuously refine safety behavior in production.

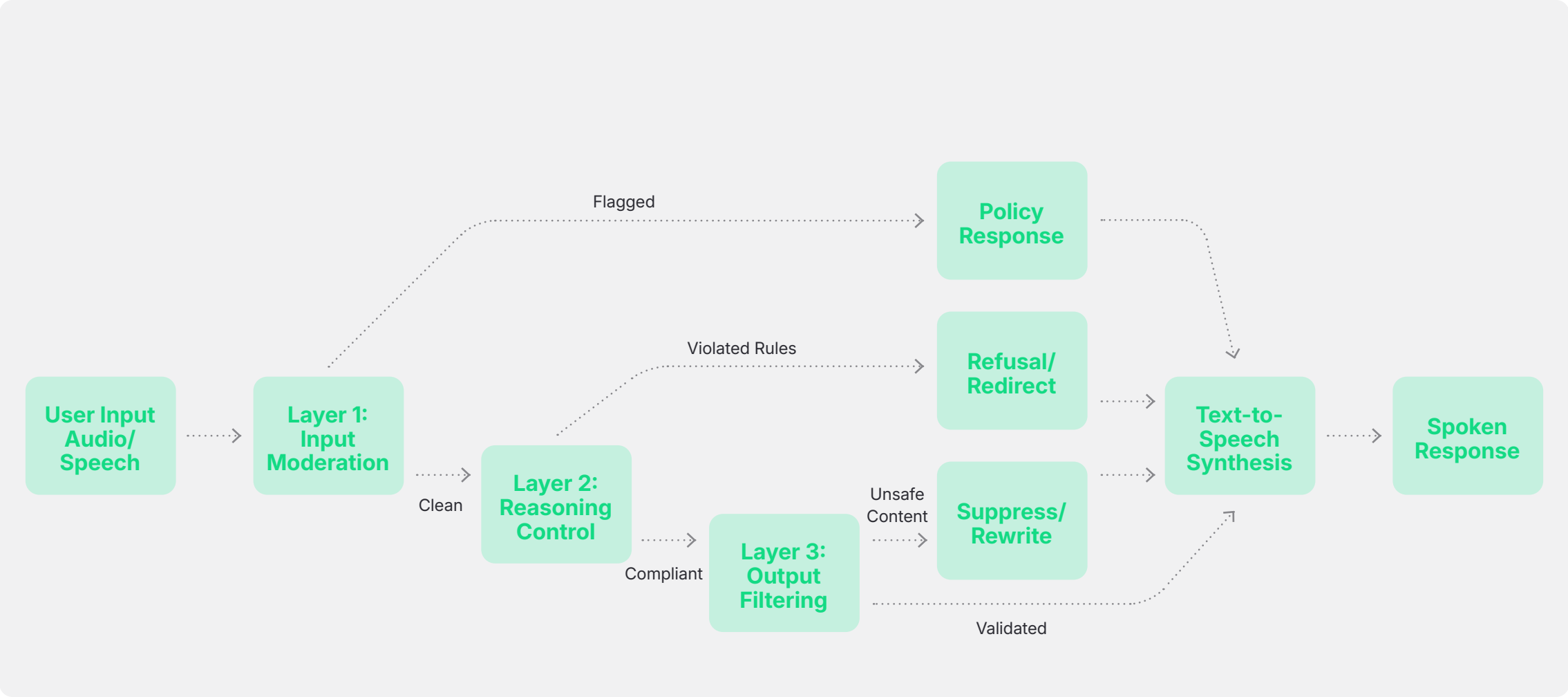


Figure 5.1: Three-Layer Guardrails Architecture — Input moderation, reasoning control, and output filtering work together to enforce behavioral boundaries at runtime.

Input Moderation: Controlling What the Model Sees

The first line of defense is filtering user input before it reaches the reasoning layer. Input moderation reduces the likelihood that the model processes content that is abusive, unsafe, or outside its intended domain.

Use moderation or classification systems to detect categories such as hate speech, explicit content, self-harm, or criminal intent. When input is flagged, route the conversation into a policy-compliant handling path rather than passing it directly to the language model.

Common patterns include:

- Abusive language: respond with a calm boundary, such as “I can help if we keep the conversation respectful.”
- Self-harm signals: provide supportive language and escalate to appropriate resources.
- Illegal or dangerous requests: refuse clearly and redirect without elaboration.

Deepgram’s speech recognition can accurately transcribe profanity or mask it when required, allowing moderation systems to operate reliably on text output. Teams typically pair this with text-based moderation models or classifiers to cover policy categories across languages and contexts.

Guardrails should be validated before deployment. Structured regression tests that probe adversarial phrasing, edge cases, and tone ensure that unsafe inputs never propagate into production conversations.

Reasoning Control: Constraining Model Behavior

Even with moderated input, language models require explicit behavioral boundaries. These constraints live in system prompts and policy instructions and should be treated as part of the agent’s safety configuration.

Effective reasoning controls include:

- Refusal rules for violent, illegal, or disallowed requests.
- Tone constraints that enforce calm, respectful responses under provocation.
- Instruction protection rules that prevent disclosure of system or developer prompts.

Prompts should be versioned, tested, and reviewed like code. High-risk applications benefit from approval workflows for prompt changes, ensuring that updates do not introduce unintended behavior.

Many production systems now adopt layered guardrail architectures, where multiple lightweight safety checks evaluate intent, policy compliance, or hallucination risk in parallel with generation. These supervisory checks can operate continuously without adding noticeable latency, preserving

conversational flow while improving control. Frameworks such as Decagon's layered real-time guardrails demonstrate how this approach scales in production voice environments.

Output Filtering: Controlling What Is Spoken

The final and most critical safeguard verifies model output before it is converted to speech. Because spoken responses cannot be retracted, only validated text should ever reach the synthesis stage.

Run each response through a post-generation filter to detect disallowed content, sensitive data, or unauthorized disclosures. If violations are detected, suppress or rewrite the response before synthesis.

Typical safeguards include:

- Redacting numeric patterns that resemble payment or identity data.
- Masking explicit or unsafe terms.
- Blocking responses that reference restricted internal data or system state.

This final verification step is essential in voice systems. Even a single unsafe utterance can undermine trust or create regulatory exposure.

Voice-Specific Safety Risks

Voice introduces safety considerations that do not exist in text-only systems. Tone, pacing, and delivery shape how users perceive intent and trustworthiness.

Key voice-specific risks include:

- Tone and empathy mismatches that escalate frustration or appear dismissive.
- Hallucinated confirmations, where the agent implies an action succeeded before it actually did.
- Pronunciation errors in TTS that unintentionally resemble offensive language

Mitigating these risks requires both technical controls and review processes. Monitoring user interruptions, sentiment shifts, and post-call feedback often surfaces issues that automated filters miss.

Safety Governance and Continuous Improvement

Safety does not end at runtime. Governance processes ensure that guardrails remain effective as models, prompts, and user behavior evolve.

Reuse existing observability pipelines to capture moderation events, refusals, and escalations, tagging them by policy category and outcome. This keeps safety telemetry integrated with operational metrics rather than siloed in separate systems.

Maintain audit logs for every safety intervention and review them regularly to identify recurring triggers or emerging patterns. In regulated environments, this is often supported by human-in-the-loop review processes.

Some organizations extend governance further with post-conversation analysis that evaluates completed interactions for policy compliance, tone, and escalation accuracy. Continuous QA systems, such as Decagon's Watchtower-style monitoring, help close the loop by feeding insights back into prompt design, moderation rules, and detection thresholds.

Guardrails must evolve over time. New forms of prompt injection, indirect requests, and policy evasion appear regularly. Moderation logic and escalation criteria should be updated accordingly.

When sessions exceed defined risk thresholds, such as repeated violations or distress signals, agents should escalate automatically to human operators.

In voice systems, safety is about guiding conversations within clear, responsible boundaries. Well-designed guardrails protect users while preserving natural dialogue, ensuring agents remain helpful, trustworthy, and appropriate as conditions change.

Operational Realities: Pitfalls and Success Factors

Building a voice agent that works is only the starting point. Sustained success depends on how the system is introduced, adopted, and measured inside an organization. Teams that succeed treat voice AI as an operational capability that evolves over time, not a one-time deployment.

Start with Focused Wins

A common failure mode is overreach. Voice agents perform best when they begin with a narrow, high-volume workflow that is repetitive and clearly scoped. Tasks like password resets, order status checks, or appointment confirmations are ideal starting points.

Klubi started with a single repeatable workflow—outbound qualification calls in Brazilian Portuguese—before expanding to post-sales support.

This narrow scope allowed them to tune domain vocabulary and noisy environment handling while replacing ~40 human SDRs with continuous AI operation handling 90%+ of conversations end-to-end.

Constrained use cases allow teams to tune vocabulary, behavior, and integrations without the noise of edge cases. They also deliver visible ROI quickly, building confidence and momentum. Most successful voice deployments started with a single domain-specific pilot before expanding into broader coverage.

Start small, prove reliability, then widen the scope deliberately.

Integrate, Then Iterate

Production voice agents sit at the intersection of telephony, authentication, backend systems, and analytics. Underestimating integration effort is a common cause of delay.

Plan for multiple tuning cycles. Prompts will evolve. Turn-taking thresholds will need adjustment. Domain vocabulary will improve with real data. Treat the initial rollout as a pilot, not a finished product. Launch to a limited audience, observe real interactions, and refine continuously before scaling.

Stability is achieved through iterative refinement based on real usage, rather than upfront completeness.

Align Humans and Manage Change

Voice agents change how people work, and adoption depends on trust.

For employees, position the agent as a tool that removes repetitive work so humans can focus on complex or high-value interactions. Involve frontline staff early. Their input improves agent behavior and creates ownership rather than resistance. Provide simple feedback loops so human agents can flag issues, suggest improvements, or review transferred calls.

For customers, transparency matters. Clearly identify the agent as AI and give it a consistent voice and identity. When users know what they are interacting with, expectations align and satisfaction improves.

Design for Hybrid Operation

The most durable deployments combine AI and humans. The agent handles what it can, and humans take over when needed.

Handoffs should preserve context. Passing transcripts or summaries prevents users from repeating themselves and reduces frustration. AI can also support human agents by summarizing prior interactions or suggesting next steps, improving efficiency without removing human judgment.

Hybrid systems work because they acknowledge limits and design for them explicitly.

Measure Incremental ROI

Return on investment builds progressively. Measure coverage, such as the percentage of calls handled end to end, alongside efficiency metrics like reduced handling time or deflection rate.

Even partial automation creates value. If an agent completes intake before transferring to a human, it still reduces workload and improves throughput. Track improvements over time and use them to guide expansion.

Progress compounds when each stage is measured and validated.

Optimize for Human Outcomes

Voice automation succeeds when it improves outcomes for people. Highlight the impact on employees and customers: fewer repetitive tasks, shorter wait times, and more consistent experiences.

For example, automating hundreds of routine calls per day translates directly into hours returned to human agents for higher-value work. Share these results internally to reinforce that the goal is service quality and empowerment, not replacement.

Sustain Momentum

Most failures are organizational, not technical. Teams that succeed adopt a crawl, walk, run mindset: start narrow, involve people, measure impact, and iterate continuously.

Celebrate incremental wins. Treat improvement as ongoing work, not a final milestone. Trust in voice AI is earned through consistent, reliable interaction over time, one practical improvement at a time.

CHAPTER 6

Applied Architectures

Reference Architectures (Topologies)

Designing a real-time voice agent is an architectural problem, not a wiring exercise. How audio, reasoning, and synthesis are composed determines conversational timing, state coherence, and reliability. Deepgram's platform supports multiple deployment topologies, from fully managed streaming loops to deeply customized, multi-service runtimes.

The following reference architectures illustrate proven patterns across that spectrum. Each reflects a production-grade implementation and highlights trade-offs between simplicity, extensibility, and operational control.

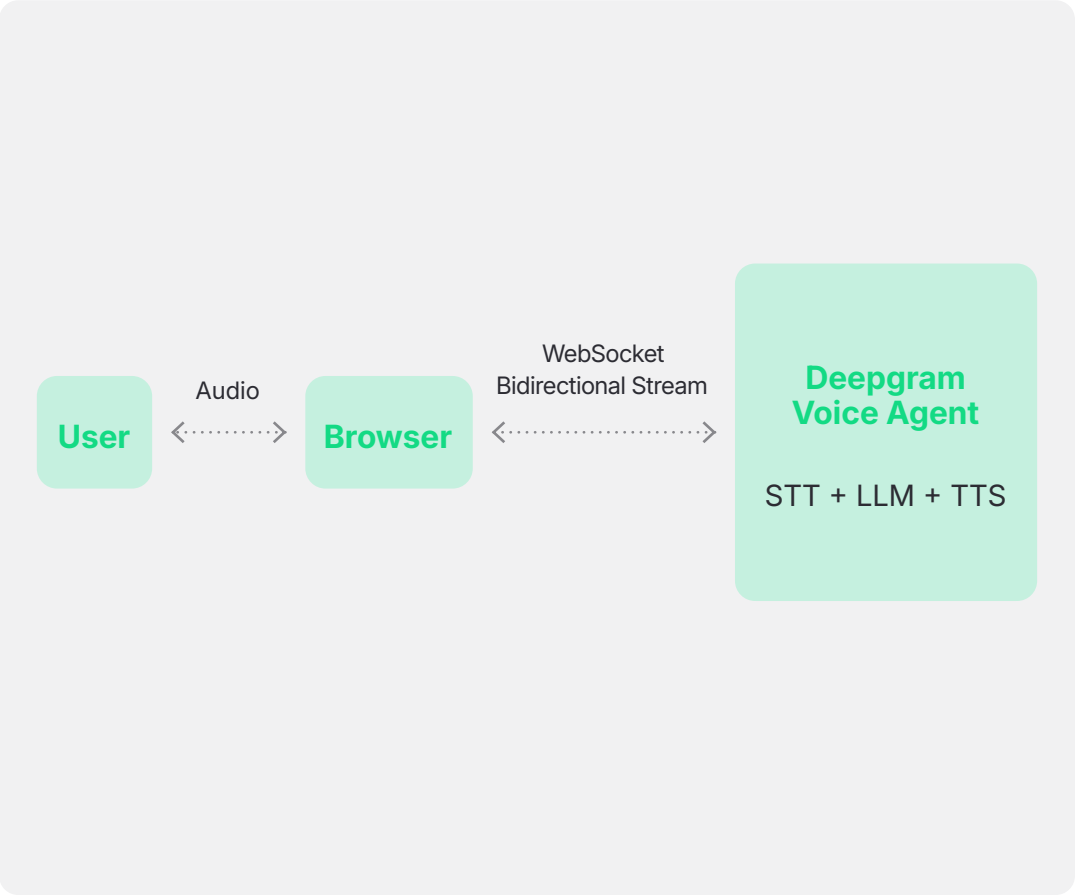
Tier 1 – Single-Agent Foundations

Baseline Voice Agent (End-to-End Streaming Loop)

Reference Implementation: GitHub

This pattern represents the simplest viable architecture for a natural, real-time voice agent: a single, persistent streaming loop that handles perception, reasoning, and speech synthesis end to end.

Topology



A lightweight browser client captures microphone audio and opens a secure WebSocket connection to the Deepgram Voice Agent API. Audio is streamed continuously. The agent runtime performs speech recognition, routes transcripts to an integrated LLM, and streams synthesized speech back using Aura-2. Playback begins immediately as audio arrives, creating a full-duplex conversational loop.

Because the entire interaction runs over one persistent WebSocket, there is no separate backend orchestrator. Conversational state, timing, interruption handling, and turn management are owned by the managed agent runtime. The client remains focused on transport, authentication, and playback, using short-lived JWTs for stateless session control.

This topology prioritizes architectural compression: fewer moving parts, fewer failure modes, and minimal latency. All conversational intelligence lives in the agent runtime, while the client acts as a thin edge.

When to use it

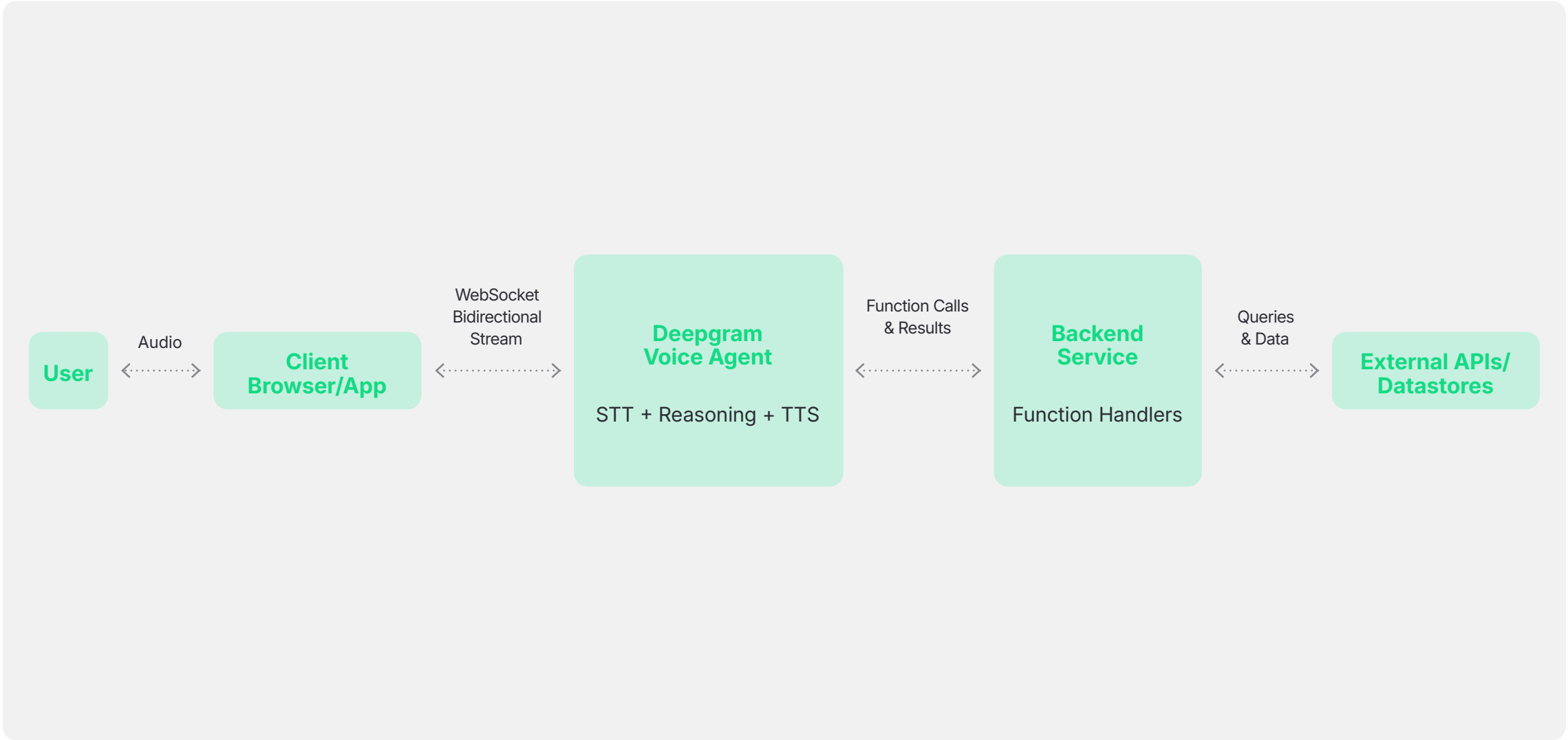
This pattern is well suited for prototypes, internal tools, and early production deployments where speed, simplicity, and conversational responsiveness matter more than deep system integration.

Function-Calling / Tool-Use Agent

Reference Implementation: GitHub

This pattern extends a real-time voice agent with external actions and data access through structured function calls.

Topology


User
Audio
Client Browser/App
WebSocket Bidirectional Stream
Deepgram Voice Agent
STT + Reasoning + TTS
Function Calls & Results
Backend Service
Function Handlers
Queries & Data
External APIs/ Datastores

Deepgram's Voice Agent API continues to manage the conversational loop, including streaming speech recognition, turn-taking, reasoning, and TTS. When an interaction requires action, the agent emits a structured function call rather than free-form text. These events are handled by a backend service (Flask in this reference implementation), which executes trusted logic and returns results to the agent for immediate verbalization.

The backend exposes a small, explicit set of functions such as customer lookup, order status, or scheduling. Each function maps directly to a deterministic handler. The demo uses mock but realistic datasets stored as timestamped JSON files to enable repeatable testing without external dependencies.

This architecture separates concerns cleanly:

- The agent runtime owns dialogue, timing, and state.
- Function definitions constrain what the model is allowed to do.
- Backend handlers execute side effects and data access.

When to use it

This topology is ideal when an agent must interact with live systems or perform transactions without breaking the voice session.

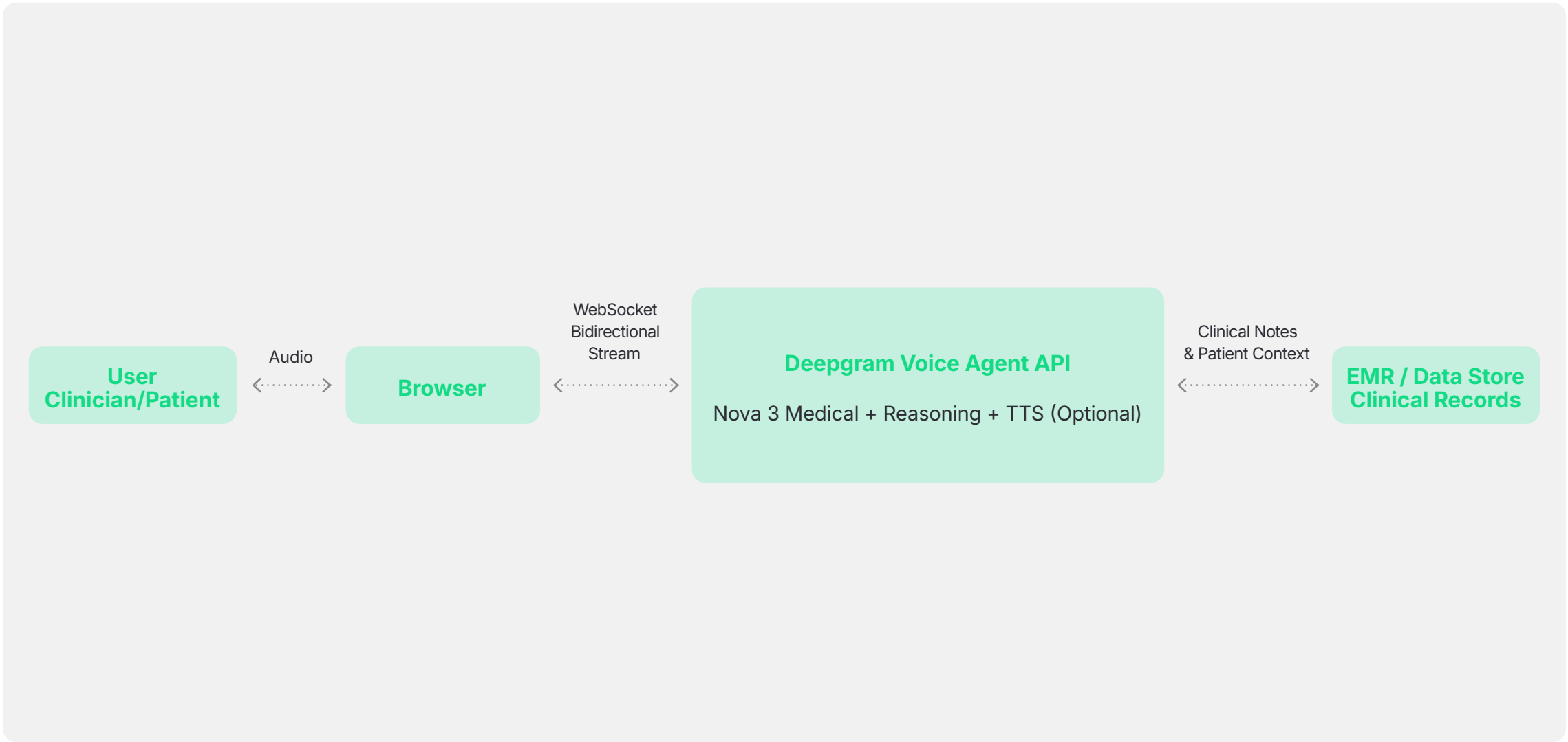
Tier 2 – Specialized and Localized Agents

Domain-Specific Voice Agent (Medical Assistant)

Reference Implementation: GitHub

This pattern shows how a general-purpose voice agent can be adapted for regulated, domain-specific workflows such as clinical documentation. It extends the baseline Voice Agent API topology with domain-optimized speech models and structured reasoning to handle specialized vocabulary, privacy constraints, and downstream data systems.

Topology


User
Clinician/Patient
Audio
Browser
WebSocket
Bidirectional
Stream
Deepgram Voice Agent API
Nova 3 Medical + Reasoning + TTS (Optional)
Clinical Notes
& Patient Context
EMR / Data Store
Clinical Records

Audio is streamed directly from the browser into the Voice Agent API, where Nova-3 Medical provides accurate transcription of clinical terminology and abbreviations. The resulting text is passed to a reasoning layer that structures the content into summaries or draft clinical notes suitable for review or export. Optional TTS can be used for confirmations or summaries, though the primary output is structured text rather than conversational reply.

The key architectural characteristic is **domain specialization without orchestration change**. The real-time speech loop remains intact, while domain intelligence is introduced through model choice, prompting, and downstream integrations. The same pattern applies to other regulated domains such as finance or legal by substituting the speech model and knowledge layer.

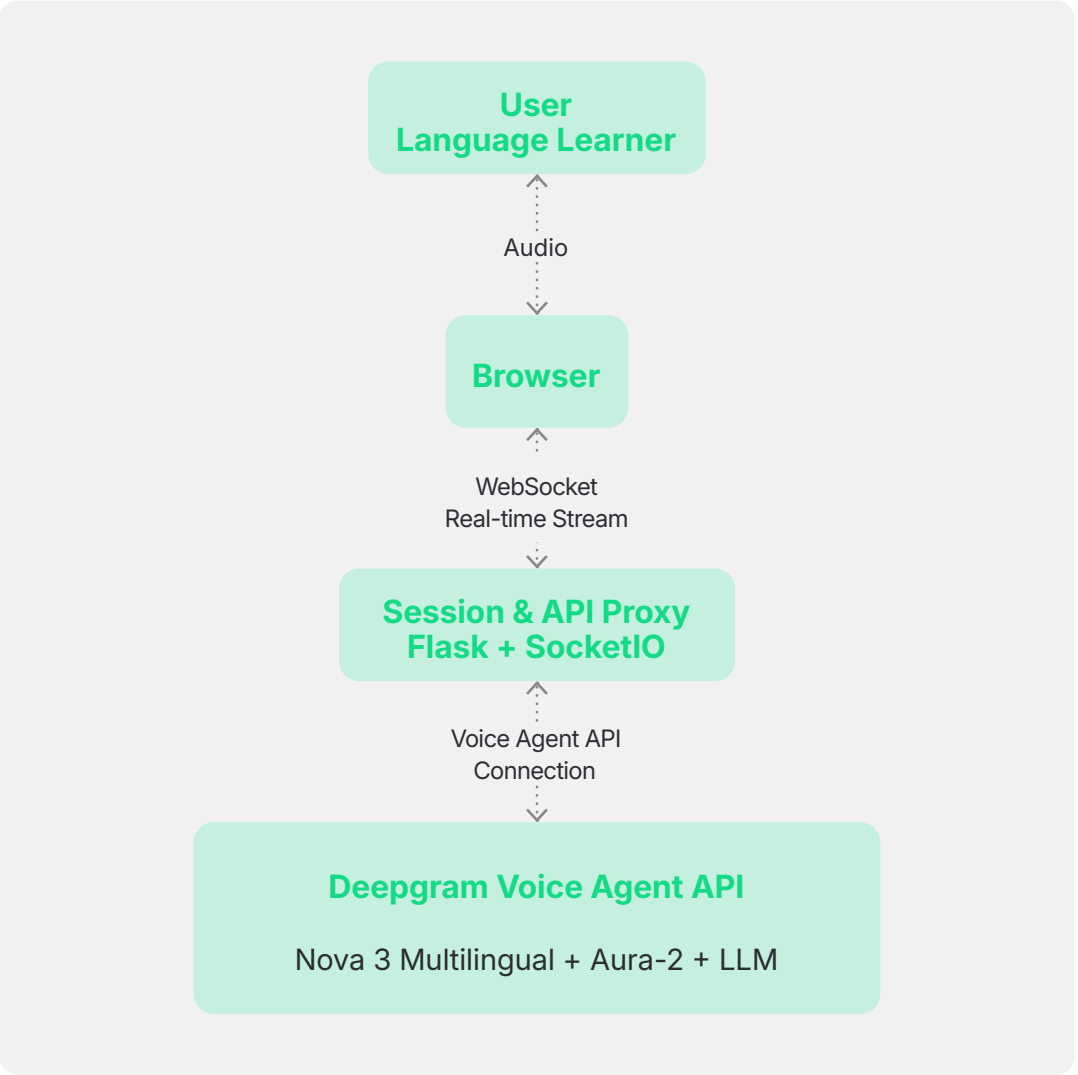
When to use

Healthcare or regulated workflows that require high transcription accuracy and structured outputs, without rebuilding the real-time speech pipeline.

Language Coach (Multilingual and Localization Pattern)

Reference Implementation: GitHub

Topology



This pattern demonstrates linguistic specialization: a multilingual, code-switching voice agent that supports real-time conversation, pronunciation feedback, and adaptive responses across languages within a single session.

A browser client streams audio through a thin backend to the Voice Agent API. Nova-3 Multilingual handles real-time transcription across languages within one continuous session, enabling natural code-switching without reinitialization. Speech output is generated via Aura-2, with voices updated dynamically as language context changes to maintain persona consistency.

The reasoning layer focuses on corrective feedback and guided conversation rather than task execution. Because multilingual STT, dynamic TTS switching, and localized prompting all operate inside the same streaming loop, the agent preserves timing and conversational flow even as languages change mid-session.

This architecture directly operationalizes the multilingual strategies described earlier: unified transcription, adaptive synthesis, and language-aware prompting coordinated in real time.

When to use

Language learning, global assistants, or cross-market applications that require seamless multilingual interaction within a single conversation.

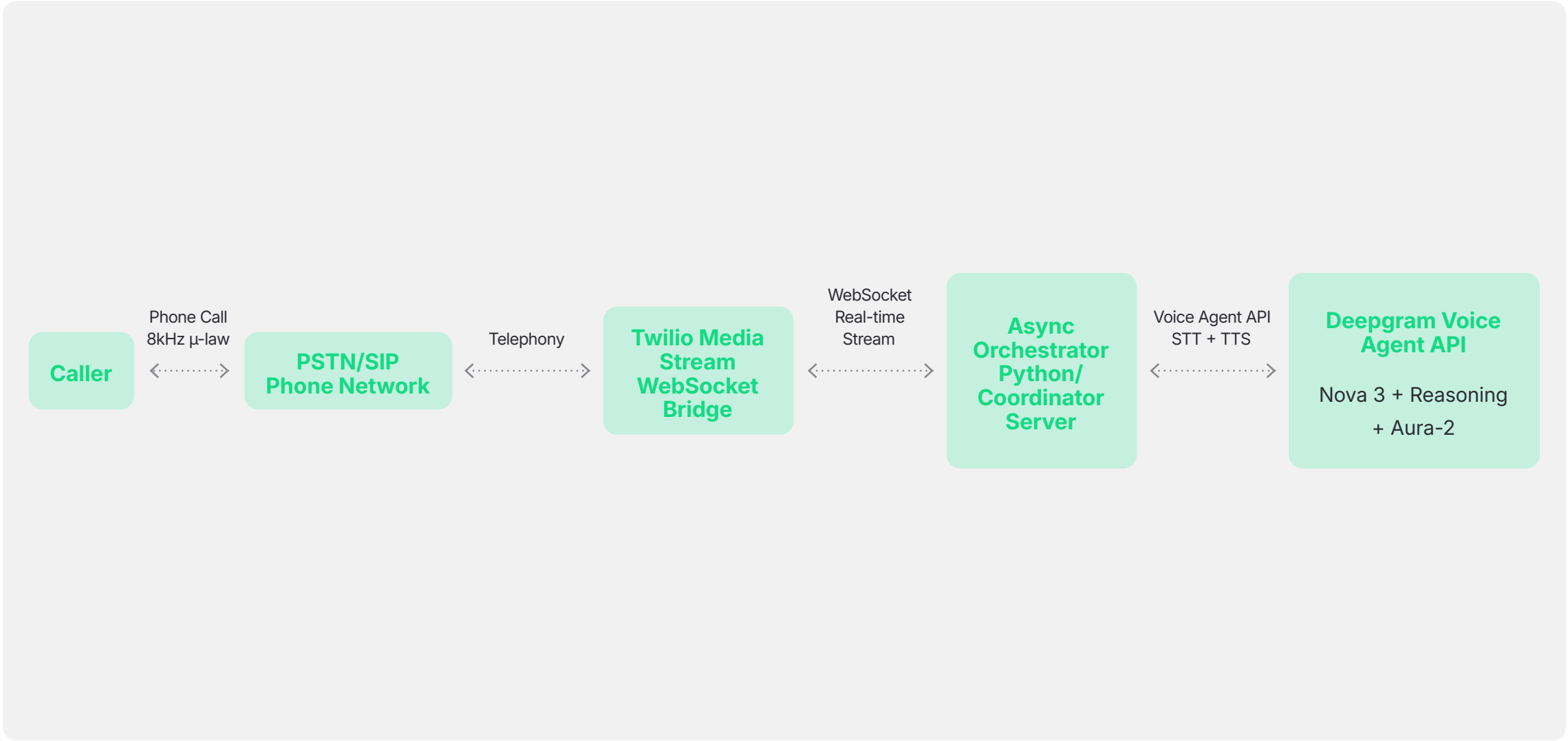
Tier 3 – Integrated and Distributed Systems

Telephony Voice Agent (PSTN / SIP Frontend)

Reference Implementation: Documentation and GitHub

This pattern connects Deepgram's Voice Agent API to traditional phone networks via Twilio Media Streams, enabling real-time, natural conversations over PSTN or SIP.

Topology


Caller
Phone Call
8kHz μ-law
PSTN/SIP
Phone Network
Telephony
Twilio Media
Stream
WebSocket
Bridge
WebSocket
Real-time
Stream
Async
Orchestrator
Python/
Coordinator
Server
Voice Agent API
STT + TTS
Deepgram Voice
Agent API
Nova 3 + Reasoning
+ Aura-2

Twilio bridges 8 kHz μ-law audio from the phone network into a bidirectional WebSocket stream. The Voice Agent API handles transcription, reasoning, and synthesis, returning Aura-2 audio that Twilio injects directly back into the live call. A small async server coordinates inbound audio, outbound playback, and interruption handling.

Key capabilities are built into the topology:

- Low-latency turn-taking with barge-in support
- DTMF handling for IVR-style interactions
- Session-level call control and clean teardown
- Secure, token-based authentication for live streams

This architecture modernizes legacy IVR and call-center systems without requiring custom telephony stacks. It preserves PSTN compatibility while enabling real-time AI interaction and observability through streamed transcripts.

When to use

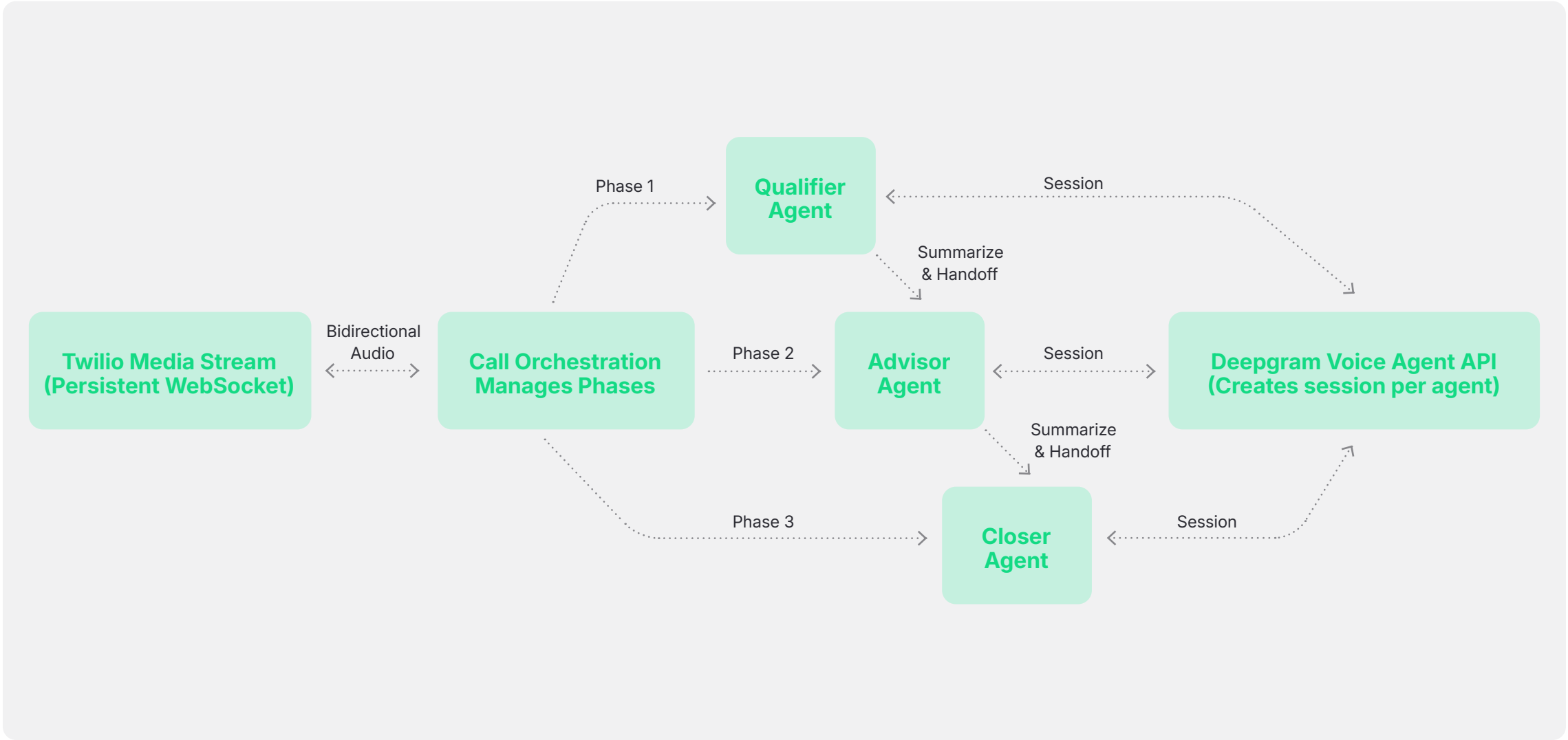
Inbound or outbound phone agents, IVR modernization, customer support automation, or agent-assist systems that must operate over standard telephone infrastructure.

Multi-Agent Orchestration (Specialized Agents with Context Handoff)

Reference Implementation: GitHub

This pattern scales a single voice agent into a coordinated system of specialized sub-agents, each responsible for a distinct phase of a conversation. A central orchestrator maintains the live audio stream while rotating agents behind the scenes.

Topology


Twilio Media Stream (Persistent WebSocket)
Bidirectional Audio
Call Orchestration Manages Phases
Phase 1
Qualifier Agent
Session
Summarize & Handoff
Phase 2
Advisor Agent
Session
Deepgram Voice Agent API (Creates session per agent)
Summarize & Handoff
Phase 3
Closer Agent
Session

A persistent Twilio WebSocket carries audio for the entire call. The orchestrator keeps this connection open while creating short-lived Voice Agent sessions for each sub-agent. Each agent runs with a focused prompt, limited toolset, and clear role, then hands off control when its task is complete.

Between phases, the orchestrator summarizes the prior exchange and injects that summary as context for the next agent. This prevents prompt bloat, isolates failures, and keeps reasoning targeted. Audio streaming continues uninterrupted across agent transitions, so the caller experiences a single, continuous conversation.

This architecture enables:

- Explicit phase separation and cleaner reasoning
- Controlled context growth through summarization
- Easier debugging and evaluation per agent role
- Flexible composition of complex workflows

When to use

Multi-step interactions such as sales qualification, onboarding, triage, or claims processing, where different conversational stages benefit from distinct prompts, logic, or success criteria.

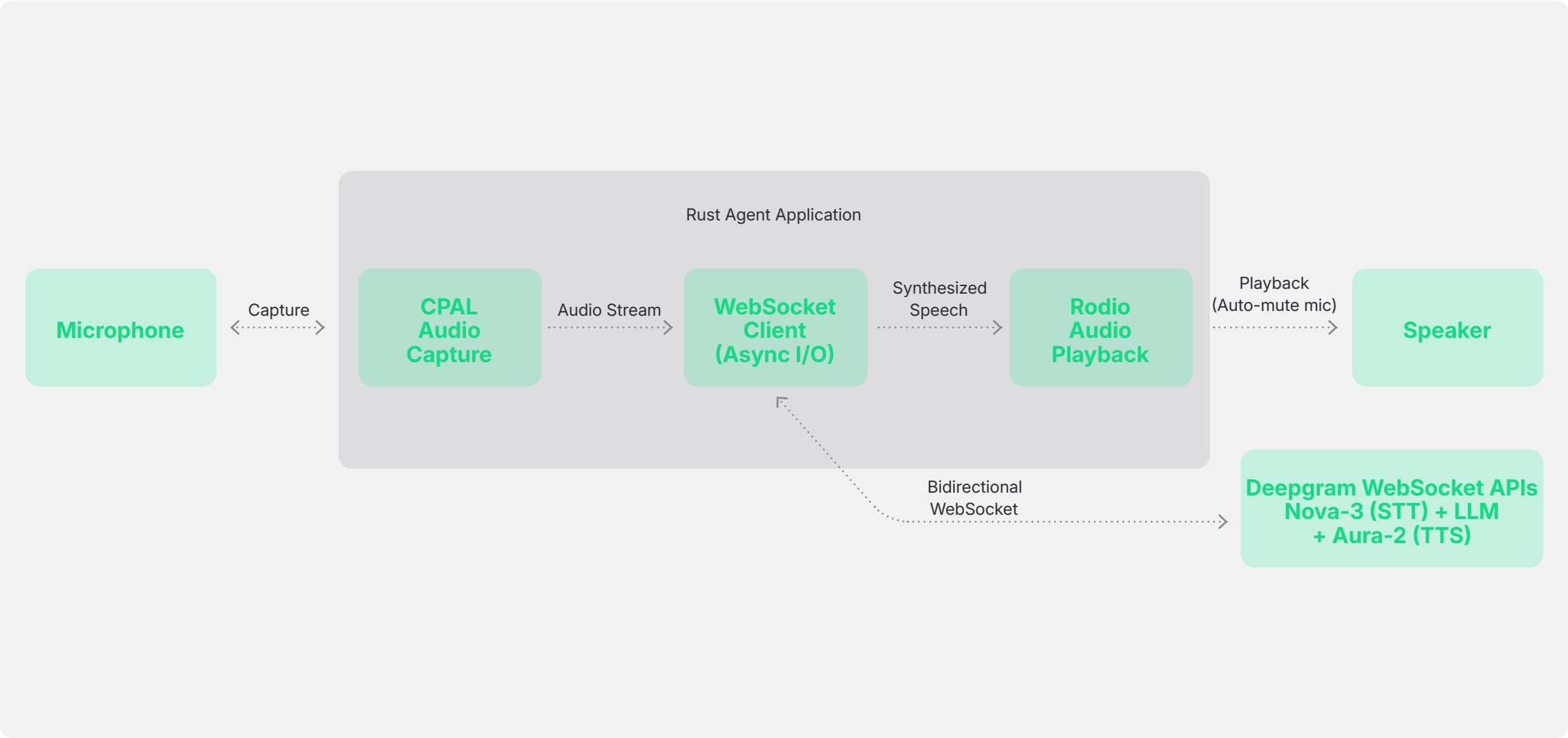
Tier 4 – Low-Level and Edge Implementations

Native SDK / Embedded Voice Agent (Rust)

Reference Implementation: GitHub

This pattern demonstrates a fully custom voice agent built directly on Deepgram's WebSocket APIs, without the managed Voice Agent runtime. Implemented in Rust, it prioritizes low latency, deterministic control, and tight integration with local audio hardware.

Topology


Rust Agent Application
Microphone
Capture
CPAL Audio Capture
Audio Stream
WebSocket Client (Async I/O)
Synthesized Speech
Rodio Audio Playback
Playback (Auto-mute mic)
Speaker
Bidirectional WebSocket
Deepgram WebSocket APIs Nova-3 (STT) + LLM + Aura-2 (TTS)

The agent streams microphone audio to Deepgram for real-time transcription and reasoning, then plays synthesized speech locally with precise timing. Audio input and output are handled asynchronously, with automatic microphone muting during playback to prevent feedback and support natural turn-taking.

Configuration is explicit and lightweight:

- **Listen:** Nova-3 (streaming STT)
- **Think:** compact reasoning model
- **Speak:** Aura-2 voice
- **Audio:** linear PCM with device-level control

This architecture exposes the full speech pipeline while avoiding unnecessary abstractions. It is well suited to environments where latency budgets are tight, dependencies must be minimal, or cloud-managed runtimes are not viable.

When to use

Edge devices, kiosks, embedded systems, IVR gateways, or on-prem deployments that require maximum control over audio I/O, threading, and runtime behavior.

Synthesis: The Architecture Continuum

These reference architectures form a practical continuum:

Tier	Focus	Example Patterns	When to Use
1	Unified simplicity	Baseline Agent, Function Calling	Prototypes and managed agents
2	Specialization	Medical Assistant, Language Coach	Domain or multilingual systems
3	Integration	Telephony, Multi-Agent Orchestration	Enterprise workflows
4	Performance & control	Native SDK / Edge Agent	On-prem and embedded environments

Most real-world deployments evolve across tiers. Teams often begin with a managed, end-to-end agent and progressively introduce custom orchestration, telephony, or edge execution as requirements grow. Deepgram's APIs and runtime model support this progression without forcing architectural resets, allowing systems to scale in complexity while preserving real-time conversational performance.

Ecosystem Patterns (Integrations)

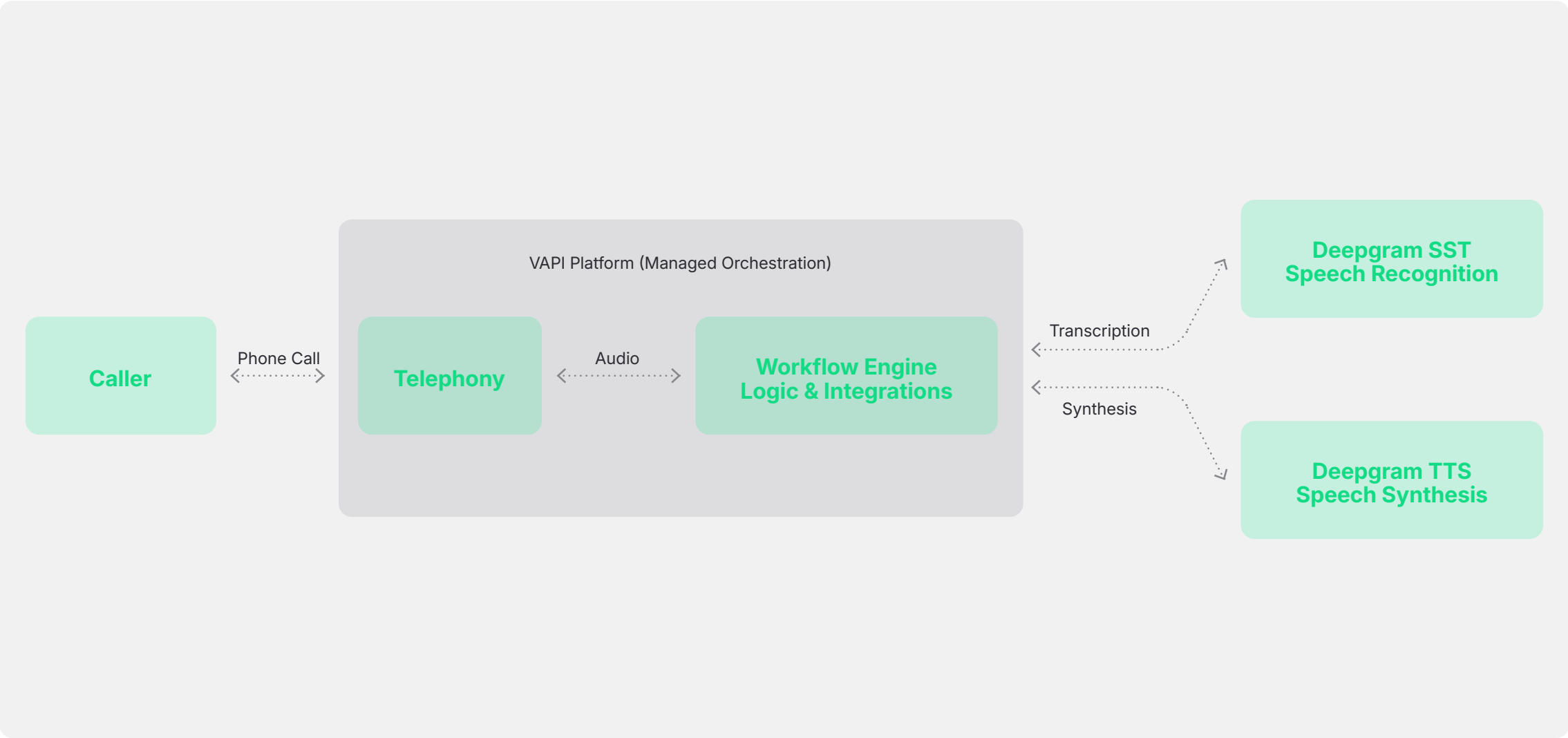
The previous section focused on reference architectures owned end to end by the builder. This section shows how those same architectural principles appear in **ecosystem integrations**, where Deepgram provides the real-time speech layer inside partner-managed orchestration, transport, or enterprise platforms.

Each pattern below represents a distinct integration topology and clarifies how responsibility is divided between Deepgram and the surrounding system.

VAPI Orchestrated Agent (Managed Platform Topology)

Reference: Vapi documentation

Topology


Caller
Phone Call
VAPI Platform (Managed Orchestration)
Telephony
Audio
Workflow Engine
Logic & Integrations
Transcription
Synthesis
Deepgram SST
Speech Recognition
Deepgram TTS
Speech Synthesis

This pattern represents a fully managed orchestration model. VAPI owns telephony, dialog flow, state management, and integrations. Deepgram supplies streaming speech recognition and synthesis as pluggable components within that workflow.

Audio from the caller is streamed to Deepgram for real-time transcription. VAPI's workflow engine interprets the text, executes logic or API calls, and then invokes Deepgram's TTS to synthesize the response. Speech is returned to the caller through the same telephony channel.

VAPI's platform demonstrates the managed orchestration model where telephony, dialog flow, and state management are abstracted, allowing teams to improve speech accuracy and latency by swapping STT/TTS components without redesigning orchestration. This topology accelerates deployment for teams prioritizing speed over deep system integration.

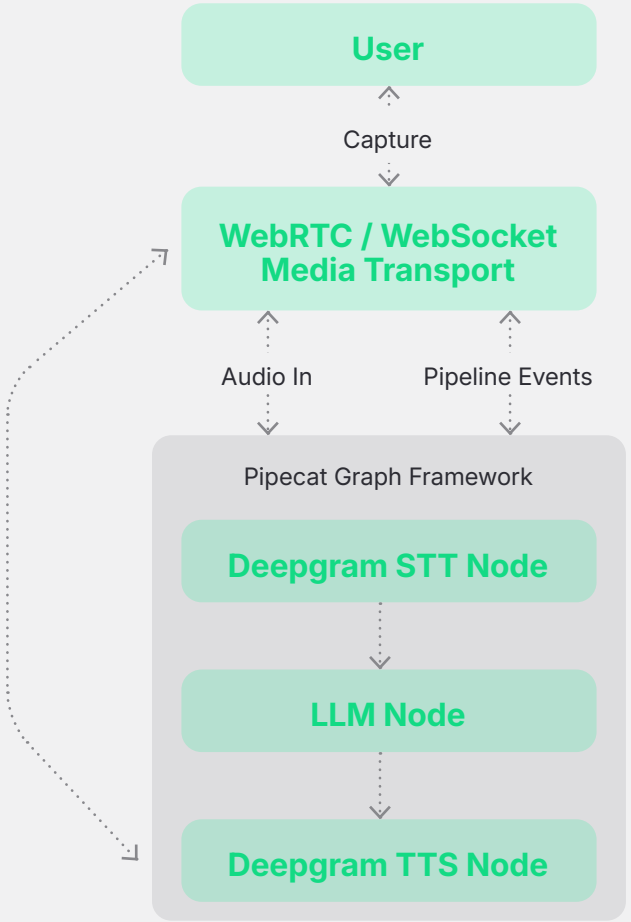
When to use it

Teams that want rapid deployment using low-code or no-code tooling, with minimal infrastructure ownership. This topology allows organizations to improve speech accuracy and latency by swapping in Deepgram without redesigning orchestration.

Pipecat Graph Agent (Open Orchestration Topology)

Reference: Pipecat documentation

Topology



This pattern uses Pipecat's open, graph-based orchestration framework. Media transport, transcription, reasoning, and synthesis are expressed as discrete nodes connected in a real-time pipeline. Deepgram appears as the STT and TTS nodes within that graph.

Developers control the full execution path: partial transcripts, interruption handling, LLM selection, and output routing. Deepgram provides the low-latency speech layer, while Pipecat manages media transport and event coordination.

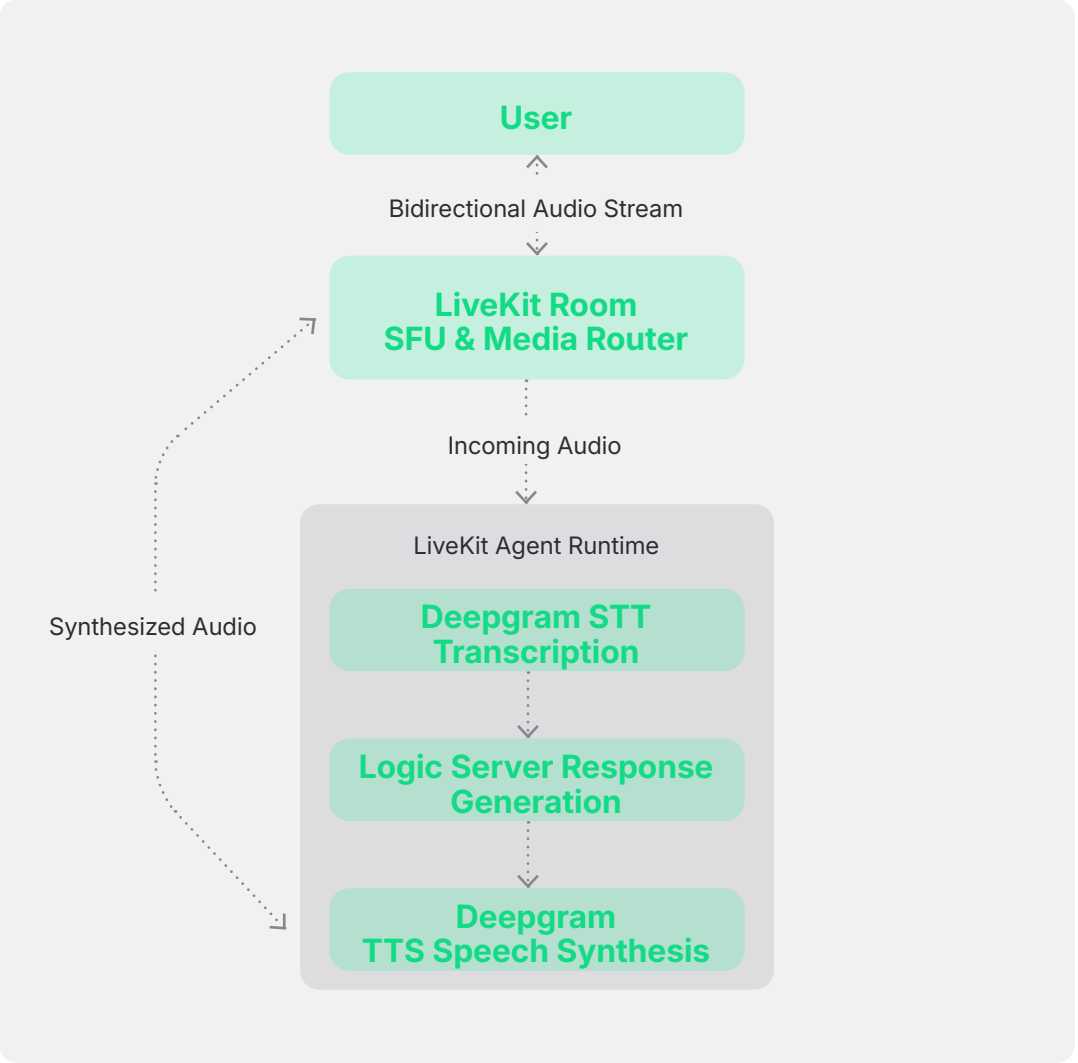
When to use it

Teams that want full transparency and customization of the voice pipeline, including experimentation, research, or advanced orchestration. Ideal for builders who want modularity without implementing media infrastructure from scratch.

LiveKit Audio Room Agent (Transport Topology)

Reference: LiveKit documentation

Topology



This pattern centers on LiveKit as the real-time media transport layer. Conversations occur inside persistent audio rooms, with the AI agent participating as a peer. Deepgram processes incoming audio for transcription and generates synthesized speech that is published back into the room.

LiveKit handles participant management, routing, and scalability. Deepgram provides speech recognition and synthesis. Application logic runs alongside the agent to determine responses or assist human participants.

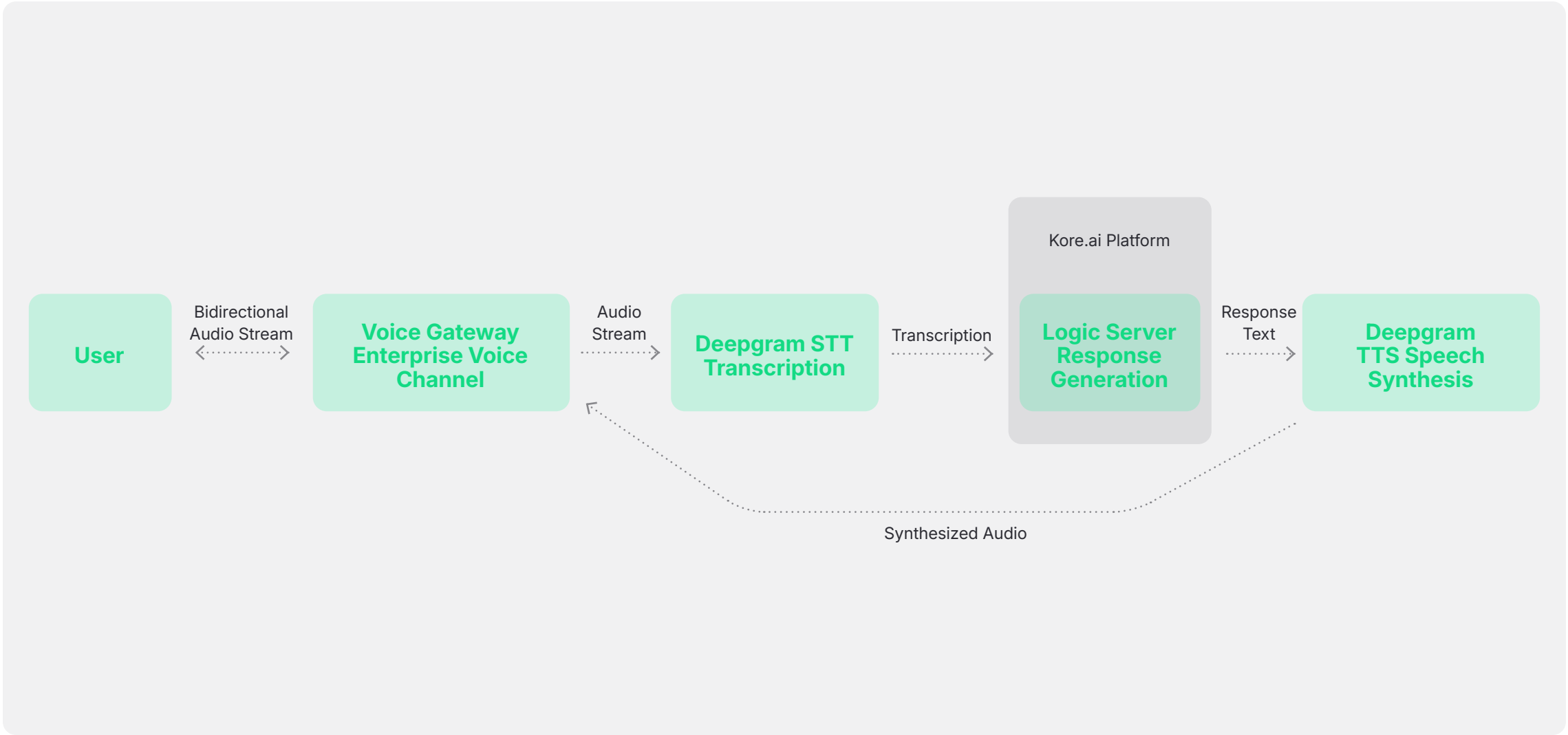
When to use it

Multi-party or interactive environments such as meetings, agent-assist, virtual rooms, or voice-enabled applications where WebRTC transport and low-latency media routing are already required.

Enterprise Conversational Platforms
(Integrated CX Topology – Kore.ai Example)

Reference: Kore.ai documentation

Topology


User
Bidirectional Audio Stream
Voice Gateway Enterprise Voice Channel
Audio Stream
Deepgram STT Transcription
Transcription
Kore.ai Platform
Logic Server Response Generation
Response Text
Deepgram TTS Speech Synthesis
Synthesized Audio

Enterprise CX platforms like Kore.ai provide full conversational ecosystems spanning design tools, analytics, compliance, and CRM integration. In this topology, the platform owns orchestration and governance, while Deepgram supplies the speech layer.

Audio is streamed from the voice gateway to Deepgram for transcription. Kore.ai's NLU and workflow engine determines the response and invokes Deepgram's TTS for synthesis. The reply is delivered through the same enterprise voice channel.

When to use it

Large organizations already standardized on an enterprise conversational platform that want to upgrade speech quality and latency without re-architecting workflows, compliance, or analytics systems.

Summary

Across these ecosystem patterns, Deepgram consistently operates as the real-time speech layer, independent of where orchestration or transport lives. Whether embedded in a managed platform, an open graph framework, a WebRTC transport, or an enterprise CX suite, Deepgram provides low-latency transcription and synthesis while adapting cleanly to different architectural control points.

These integrations show how Deepgram's APIs fit naturally at multiple layers of the voice stack, enabling teams to improve speech performance without constraining how the rest of the system is designed.

CHAPTER 7

The Future of Voice AI

The Next Architectural Shift

Voice agents today are built on a modular pipeline: speech-to-text, text-based reasoning, then text-to-speech. This architecture has scaled well, but it introduces structural inefficiencies. Audio is repeatedly compressed into text and expanded back into speech, discarding expressive signals such as tone, pacing, and emotion along the way.

Deepgram's Neuroplex research explores the next architectural shift: **speech-to-speech (S2S) systems** that listen, reason, and respond directly in audio. Instead of treating text as the primary interface between perception and generation, Neuroplex operates on continuous representations of speech, preserving meaning beyond words alone.

In a traditional pipeline, the phrase "I guess so" produces the same transcript regardless of whether it is spoken hesitantly, sarcastically, or enthusiastically. A speech-native architecture retains those distinctions through the reasoning step, allowing responses to reflect emotional and conversational context rather than just lexical meaning.

Neuroplex is designed around this principle. Internally, it operates on dense semantic and acoustic representations rather than symbolic transcripts. Input audio is mapped to output audio through an internal understanding layer, enabling the system to reason over prosody, emphasis, and timing in ways that text pipelines cannot.

This approach enables several structural advantages:

- **Lower latency**: Fewer handoffs allow the system to begin formulating responses while still listening, supporting overlap and natural turn-taking.
- **Contextual robustness:** Meaning is refined dynamically rather than committed early to a fixed transcript.
- **More natural interaction**: Interruptions, backchannels ("mm-hmm"), and partial utterances are modeled as learned behavior rather than hard-coded rules.

Early S2S systems have demonstrated promise, but many struggle in production. Common issues include opaque behavior, limited steerability, policy violations, and difficulty debugging failures. Neuroplex addresses these constraints by combining end-to-end continuity with modular control.

"Hello" 15,000 times
in Current Architectures

"Hello" 15,000 times
in Neuroplex Architectures

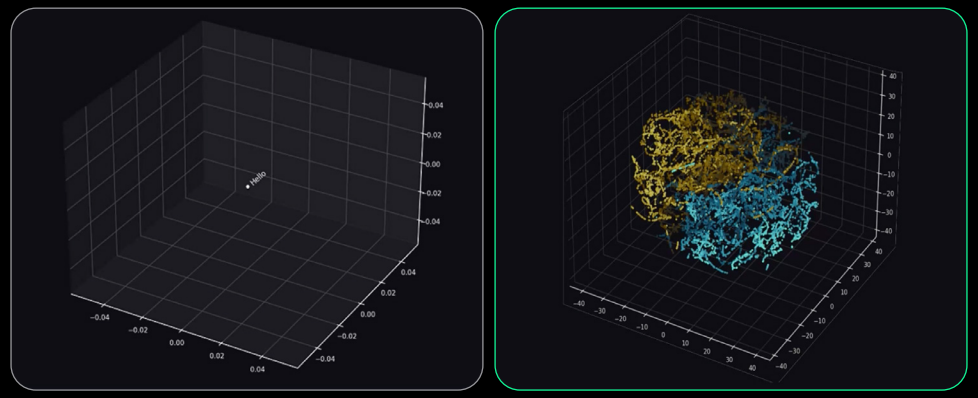


Figure 1: Different acoustic realizations of the same word occupy distinct regions in Neuroplex's latent space, preserving nuance that text collapses into a single token.

Neuroplex Architecture Overview

- Neuroplex is **end-to-end trainable but modular by design**.
  It preserves speech continuity while exposing control points required for production systems.

Core architectural elements include:

- **Adapter-based composition**
  Speech encoders, reasoning models, and speech decoders are connected via learned adapters. These adapters translate internal representations without emitting intermediate text, preserving acoustic and semantic features across modules.
- **Continuous latent flow**
  Conversations exist as dense vector streams that carry prosody, emphasis, and emotional tone across turns.
- **Inspectable internals**
  Each module can emit debug artifacts, enabling developers to inspect latent states, reasoning signals, or alignment decisions (capabilities largely absent from black-box S2S models).

- **Steerable generation**
  Despite operating end-to-end, Neuroplex supports structured guidance for tone, intent, and policy compliance through conditioning interfaces rather than brittle prompt hacks.
- **System-level optimization**
  The training objective targets end-to-end conversational quality rather than isolated metrics like WER or MOS, aligning perception and generation as a single system.

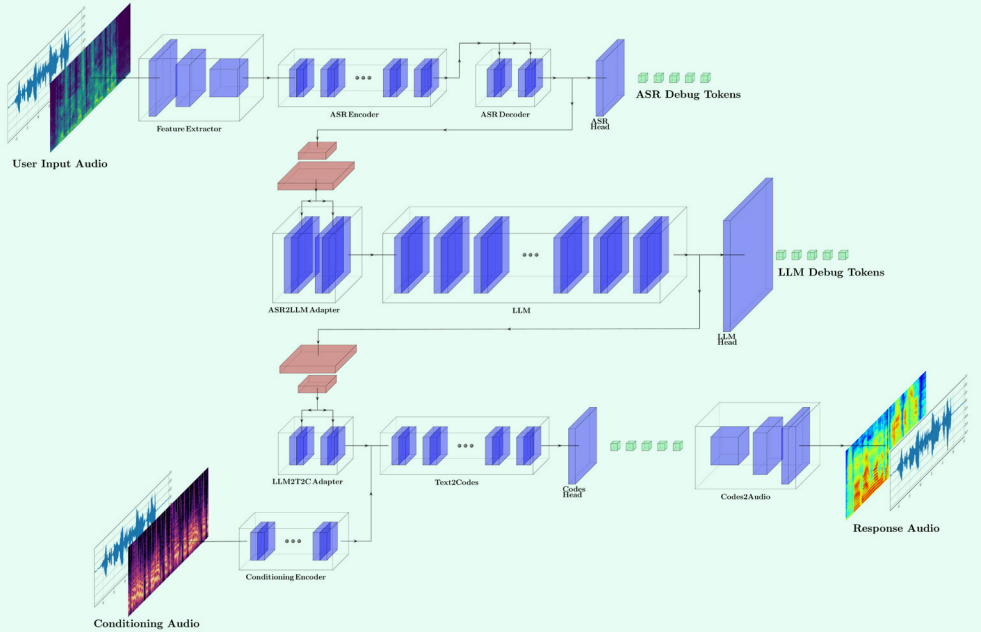


Figure 2: Neuroplex architecture showing the modular pipeline from input audio to response audio. The system consists of specialized components (Feature Extractor, ASR, LLM, Text2Codes, Codes2Audio) connected by learned adapters (ASR2LLM, LLM2T2C). Debug tokens can be extracted at multiple stages for model inspection.

Challenge in Current S2S Systems	Neuroplex Design Response
Latency grows with context size	Prior context is abstracted; only the active turn runs at full resolution
Weak instruction following	Full-scale reasoning models remain in the loop via adapters
Unpredictable speech output	Acoustic decoding is conditioned and controllable
Opaque failures	Internal states and transitions are inspectable
High compute cost	Specialized components handle distinct workloads

Neuroplex is a speech-native framework that balances fluidity with control, enabling continuous listening, thinking, and speaking without sacrificing enterprise-grade reliability.

Implications for Builders

As S2S architectures mature, the developer experience for voice agents will shift in several ways:

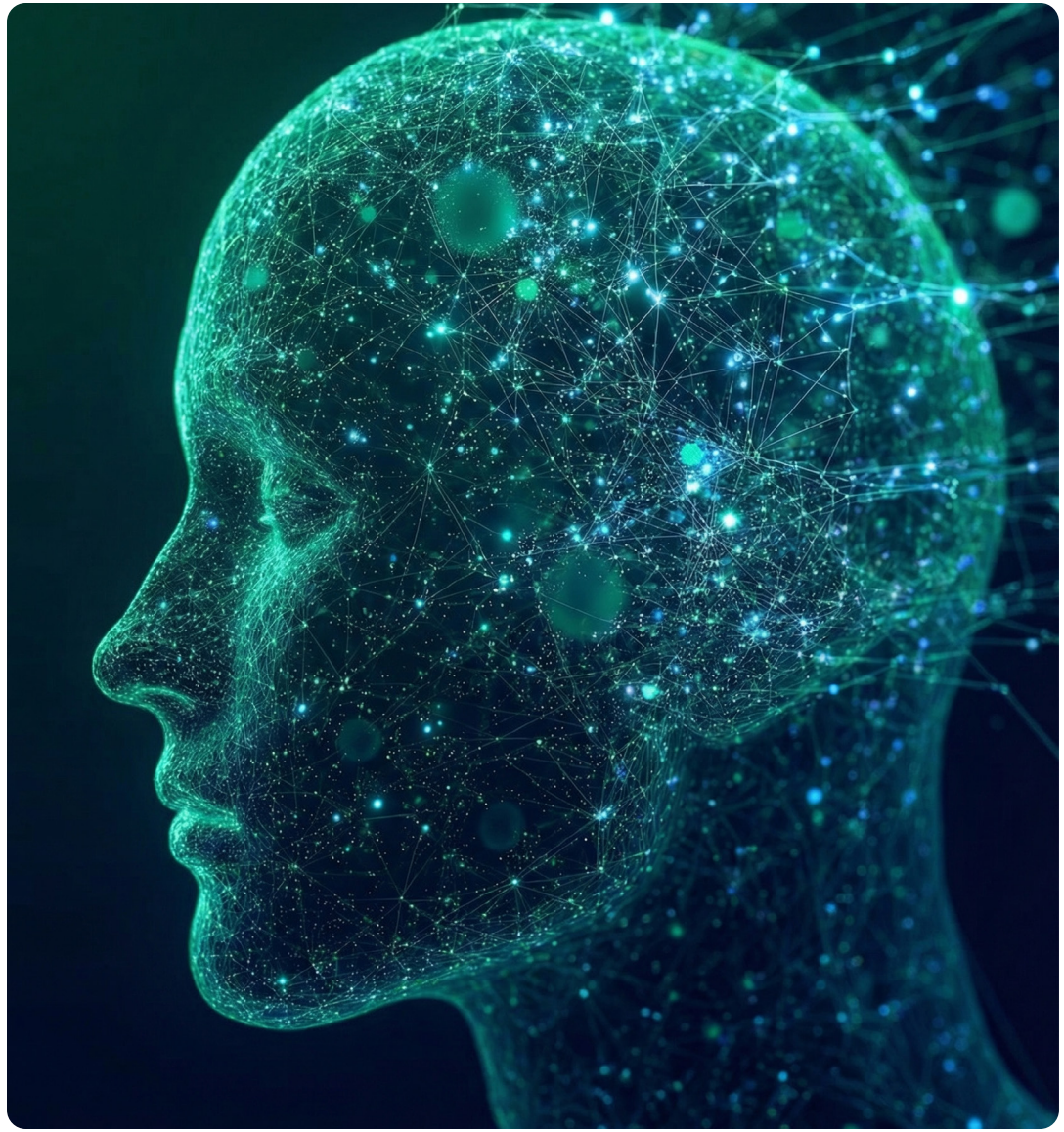
- **Audio-to-audio abstractions**
  Developers may no longer manage transcripts as the primary interface. Audio streams enter the system and audio streams return, while control logic focuses on knowledge, actions, and constraints rather than plumbing.
- **Reduced orchestration complexity**
  Language switching, barge-in handling, and turn detection can be learned behaviors rather than explicit logic, reducing integration surface area and failure modes.
- **Richer persona control**
  Acoustic style vectors enable consistent tone and affect, such as maintaining calm authority or empathetic reassurance, without embedding fragile instructions in prompts.
- **New evaluation methods**
  Perceptual listening tests and conversational quality metrics will complement logs and transcripts as primary evaluation tools.

- **Evolving deployment models**
  S2S systems are compute-intensive and will initially favor cloud or specialized accelerators. Traditional ASR will remain appropriate for lightweight tasks, while interactive agents benefit most from speech-native reasoning.
- **Converging skill sets**
  Teams previously split across ASR, TTS, and dialogue design will increasingly work within unified speech model workflows that span acoustic and semantic optimization.

Neuroplex reflects a broader transition in voice AI: moving beyond text as the organizing abstraction for conversation. Rather than removing developer control, it redefines it by aligning system design more closely with how humans listen, interpret, and respond in real dialogue.

The long-term goal is clear: approach the Audio Turing Test, where machine and human conversation become indistinguishable. Neuroplex is Deepgram's architectural blueprint for that future.

Learn more: Read the full Neuroplex technical whitepaper for detailed architecture, benchmarks, and research findings.



CHAPTER 8

Getting Started

Recap: A Practical Framework for Voice Agents

This guide has focused on how modern voice agents are designed, deployed, and operated in real production environments. The final step is turning that understanding into a working system.

Voice agents are becoming a core interface for customer support, internal automation, and real-time assistance because speech is fast, natural, and increasingly reliable at scale.

Throughout this guide, we established a clear framework:

- **Foundations:** A voice agent operates as a continuous loop of listening, understanding, reasoning, and speaking. High-quality perception (streaming STT), fast reasoning, and low-latency synthesis must work together to feel conversational.
- **System design**: Natural interaction depends on turn-taking, interruption handling, multi-turn context, tool use, telephony integration, and multilingual support.
- **Operational excellence**: Production agents require reliability testing, observability, data controls, and safety guardrails to remain performant and trustworthy over time.
- **Applied architectures**: Real-world deployments span managed agents, function-calling systems, telephony integrations, multi-agent orchestration, and edge runtimes.
- **The future:** Speech-native architectures like Neuroplex point toward end-to-end voice intelligence, where listening, reasoning, and speaking converge into a single loop.

The central takeaway: **real-time voice agents are now buildable without custom infrastructure**. Platforms like Deepgram abstract the hardest problems in streaming speech, allowing teams to start small and evolve toward more advanced architectures as requirements grow.

Deepgram's role is intentionally infrastructural. We provide fast, accurate perception, natural synthesis, and a unified runtime through the Voice Agent API, with deployment options ranging from cloud to dedicated and on-prem. You bring the logic, workflows, and experience design.

Choosing Your Build Path

Different teams start from different places. The path forward depends on your goals, constraints, and appetite for control.

For Developers

Start with a minimal, working loop. **Sign up for a free account** to create an API key, then try the **Voice Agent Playground** to test a live agent in your browser. When ready to build, follow the **Voice Agent API quickstart** to stream audio and hear responses in real time. Explore **reference implementations** such as the baseline agent, function-calling demos, or multi-agent prototypes to understand how perception, reasoning, and synthesis interact under streaming conditions. Begin with a single task, then layer in tools, memory, or telephony once the fundamentals feel intuitive.

For Architects and Product Leaders

Identify a focused workflow where voice delivers immediate value, such as scheduling, intake, or support deflection. Use the architectural patterns in this guide to define system boundaries, latency targets, and success metrics. Early attention to compliance, routing, and scale prevents rework later. Pilot narrowly, validate outcomes, then expand deliberately. **Contact our team** to discuss your deployment strategy or **explore pricing options**.

For Enterprise Evaluators

Voice agents should integrate into existing CX and analytics ecosystems, not replace them. Deepgram works across telephony, WebRTC, open orchestration frameworks, and enterprise conversational platforms, allowing teams to modernize voice interactions without re-architecting upstream systems.

Most organizations begin with a proof of concept, then scale after validating performance, cost, and governance requirements. **Contact us** to plan your enterprise POC.

Your Open Build Path

No matter how you choose to start, you are not locked into a single vendor, abstraction level, or architecture. Deepgram's ecosystem is designed for modularity and evolution. Teams often begin with a managed platform to validate experience and latency, then migrate to the Voice Agent API for deeper customization, or build directly on STT and TTS for maximum control.

Across these paths, Deepgram serves as the consistent real-time speech layer. You can integrate with **VAPI** for rapid prototyping, **Pipecat** for open orchestration graphs, **LiveKit** for real-time audio transport, or enterprise CX platforms for production workflows. The architecture is yours to design. Deepgram provides the streaming speech intelligence that makes it reliable, low latency, and production-ready.

Take the First Steps

Start small and make it tangible. The fastest way to build intuition is to interact with a live system:

Try it now:

- Voice Agent Playground – Test a complete voice agent in your browser
- Flux Playground – Experience conversational turn-taking
- Aura-2 Playground – Hear expressive text-to-speech synthesis

Once you've experienced the system live, spin up a sample application, adapt it to your domain, add a single tool or function call, and expand from there.

When you are ready to go deeper, request a design review or deployment consultation. Whether you are validating a prototype, planning a telephony rollout, or evaluating Dedicated or region-specific deployments, Deepgram provides the documentation, architectural guidance, and infrastructure to support each stage.

Voice AI has reached a point where natural, low-latency conversation is no longer speculative. The remaining work is architectural and experiential: shaping timing, behavior, and trust in real environments. You now have the frameworks to do that well.

The guide may end here, but the system you build does not. Start with one workflow. Iterate deliberately. Evolve the architecture as requirements grow.

Join the Deepgram community for ongoing support, technical discussions, and to connect with other builders.

Happy building. We're excited to see what you build as you shape the future of voice AI with Deepgram.

CHAPTER 9

Appendices

The appendices provide supporting reference material: a glossary of key terms, API endpoints and parameters for common Deepgram features, and diagnostic guidance to help identify the architectural layer responsible for common voice agent failures.

Glossary of Key Terms

ASR (Automatic Speech Recognition)
Technology that converts spoken audio into text, also referred to as STT (Speech-to-Text). Deepgram ASR models include Nova-3 for multilingual transcription and Flux for conversational speech with rapid turn detection.

CSR (Conversational Speech Recognition)
A dialogue-optimized form of ASR that supports partial results, barge-in detection, and accurate end-of-turn identification during natural conversation. Deepgram's Flux is the primary implementation of CSR in the platform.

LLM (Large Language Model)
A Transformer-based model that performs reasoning in a voice agent, interpreting transcribed speech and determining the next conversational action or response.

TTS (Text-to-Speech)
Technology that synthesizes spoken audio from text. Deepgram's Aura-2 provides real-time speech generation with multiple voice styles and languages.

Deepgram Voice Agent API
A unified WebSocket API that manages the real-time conversational loop, including streaming STT, integrated LLM reasoning, and streaming TTS playback, with support for function calls. It exposes event signals and accepts runtime configuration updates during a session. Serves as both the speech layer and orchestration runtime, eliminating the need for custom real-time coordination logic. View API Reference

Barge-in
The ability for a user to interrupt the agent while it is speaking, requiring immediate suppression or stopping of TTS output to maintain natural turn-taking.

Cascade Architecture
The traditional voice agent architecture that converts speech→text→reasoning→text→speech sequentially. Contrasts with speech-to-speech (S2S) systems like Neuroplex that operate on continuous audio representations.

Latency (Response Latency)
The elapsed time between the end of a user's utterance and the start of the agent's spoken reply, spanning ASR, reasoning, and TTS stages.

VAQI (Voice Agent Quality Index)
A composite metric proposed by Deepgram to evaluate conversational smoothness, incorporating latency, interruption handling, and missed-response rates.

WebSocket
A protocol for full-duplex communication over a single TCP connection, enabling bidirectional audio streaming between clients and voice agent services in real time.

Function Call (LLM Function Calling)
A mechanism by which an LLM emits structured JSON to request an external action, such as a database lookup, before continuing the conversation with updated context.

Full-Duplex
Simultaneous bidirectional communication allowing the voice agent to receive user audio while playing agent speech. Required for natural barge-in and interruption handling.

IVR (Interactive Voice Response)
Traditional menu-driven phone systems using prerecorded prompts and DTMF input. Modern voice agents replace or modernize IVR systems with natural language interaction.

Neuroplex
Deepgram's research architecture for speech-to-speech intelligence, connecting ASR, LLM, and TTS through a shared latent representation to preserve acoustic nuance and conversational expressiveness.

Dedicated Deployment
A single-tenant instance of Deepgram's platform deployed in a customer-controlled environment to meet security, compliance, or regional governance requirements.

DTMF (Dual-Tone Multi-Frequency)
Keypad tones generated when users press phone keys. Voice agents should detect and handle DTMF separately from speech to avoid transcript pollution.

EagerEndOfTurn / TurnResumed
EagerEndOfTurn is a medium-confidence signal from Flux that the user may be finished speaking, enabling speculative reasoning. TurnResumed indicates the user continued speaking after an eager end, triggering cancellation of speculative work.

Event-Driven Architecture
An architectural pattern where system components react to asynchronous events (e.g., speech start, turn end, interruption) rather than polling state. Essential for responsive voice agent behavior.

Redaction (PII Redaction)
The masking or removal of sensitive information from transcripts, such as payment data or personal identifiers, to support privacy and regulatory compliance.

Partial vs Final Transcript
Partial transcripts are interim ASR outputs produced while the user is still speaking. Final transcripts are confirmed at the end of an utterance. Also referred to as “interim” and “final” results in some speech recognition systems. Partial results enable earlier reasoning and lower perceived latency.

PSTN (Public Switched Telephone Network)
The traditional circuit-switched telephone network. Voice agents integrate with PSTN through telephony gateways to handle standard phone calls at 8 kHz audio quality.

Turn (Conversation Turn)
A single exchange in a dialogue, typically consisting of a user utterance followed by an agent response, coordinated through speech-boundary events.

STT vs ASR
Both refer to speech-to-text conversion. ASR is the broader technical term, while STT describes the functional capability.

Streaming vs Batch
Streaming processes audio incrementally and emits results in real time. Batch processes complete audio after recording ends. Voice agents depend on streaming operation, as batch processing introduces unacceptable latency for conversational UX.

Speech-to-Speech (S2S)
An emerging architecture that processes audio directly without text intermediaries, preserving prosody and emotional context. Deepgram’s Neuroplex represents this next-generation approach.

Utterance
A continuous segment of speech from one speaker, ending when the speaker yields the conversational turn.

NLU (Natural Language Understanding)
Traditionally refers to intent and entity extraction. In modern voice agents, LLM-based reasoning often replaces standalone NLU, though hybrid systems may still incorporate both.

Orchestration
The coordination layer that manages timing, state, and data flow between speech recognition, reasoning, and synthesis components in real time. Critical for maintaining conversational rhythm and handling interruptions.

This glossary is intended as a reference for terminology used throughout this guide.

Quick Reference: Deepgram APIs and SDKs

This section summarizes the Deepgram APIs and features most commonly used when building real-time voice agents.

Realtime Speech-to-Text (STT)

Realtime Streaming STT (Non-Flux)

Endpoint: `/v1/listen`

Low-latency streaming STT over WebSocket or SDKs. Emits interim and final transcripts with optional word timing and diarization.

Typical use: Live transcription, telephony, analytics, agent observability.

Common params:
model=nova-3, language, encoding, sample_rate, punctuate, diarize, smart_format

Conversational STT – Flux

Endpoint: `/v2/listen`

Conversational STT optimized for fast, reliable end-of-turn detection and natural pauses.

Try it: Flux Playground

Typical use: Voice agents with barge-in, natural turn-taking, and fast response loops.

When to use Flux: Use Flux (`/v2/listen`) for real-time voice agents that require conversational turn-taking. Use regular streaming STT (`/v1/listen`) for non-conversational use cases like live transcription, analytics, or when implementing custom turn detection.

Common params:
model=flux, language, encoding, sample_rate, eot_threshold, eager_eot_threshold, eot_timeout_ms

Flux Events:

- StartOfTurn - user begins speaking for the first time in the turn
- Update - periodic transcript updates
- EagerEndOfTurn — medium-confidence end (speculative)
- TurnResumed — user continued after eager end
- EndOfTurn — high-confidence end of user speech

Text-to-Speech (TTS)

Text-to-Speech API

Endpoint: `/v1/speak`

Generates audio from text using Aura voices. Supports streaming and non-streaming output.

Try it: Aura-2 Playground

Common params:
model=<model-voice>, encoding=<audio-encoding>, sample_rate=<Hz>

Voice Selection:

- **TTS:** voice=<voice_id>
- **Voice Agent:** agent.speak.voice=<voice_id>

Voice Agent API

Unified Voice Agent API

Endpoint: `/v1/agent/converse`

Single WebSocket API combining **STT + LLM reasoning + TTS**. Manages turn-taking, events, prompts, and function calls.

Try it: Voice Agent Playground

Key Event Types (Examples)

- Welcome – session initialized
- SettingsApplied – configuration active
- UserStartedSpeaking – speech detected
- ConversationText – user or agent text
- AgentThinking – LLM processing
- AgentStartedSpeaking / AgentAudioDone – agent audio lifecycle
- FunctionCallRequest / FunctionCallResponse – tool execution
- AgentWarning / AgentError – runtime issues

Runtime Updates

Supports **mid-call updates** to models, prompts, voices, and tools via configuration events (no reconnect required).

SDKs and Auth

Deepgram SDKs

Languages: Python, Node.js, .NET, Go, others

Simplify streaming, WebSocket handling, auth, reconnects, audio I/O, and event parsing.

Temporary Auth Tokens

Short-lived JWTs created via project/key endpoints.

Recommended for: Browser and streaming clients instead of long-lived API keys.

Benefits: Scoped permissions and limited TTLs.

Compliance and Language Features

Redaction & Profanity Filtering

Transcript-level masking at the API layer.

Examples:
redact=pii, profanity_filter=true

Multilingual Support

- Explicit language selection (language=en)
- Automatic detection (detect_language=true)
- Multilingual models (e.g., Nova-3 Multilingual)

Function Calling (Voice Agent API)

Structured event-based mechanism for invoking external tools.

Flow:
FunctionCallRequest → your app executes tool → FunctionCallResponse → agent continues with result

Common Failure Modes in Real-Time Voice Agents

Even well-architected voice agents can exhibit issues once deployed in real-world conditions. Because voice agents operate as tightly coupled, real-time systems, problems often emerge at the boundaries between perception, reasoning, synthesis, and transport. This section outlines the most common failure modes and, more importantly, **where in the system they typically originate**.

The goal is not to provide exhaustive fixes, but to help teams quickly narrow the scope of investigation.

Slow or Awkward Responses

Usually originates in:
End-of-speech detection, reasoning latency, or client-side audio playback.

What to inspect:
Turn-boundary signals, the timing between final user speech and agent response, and whether audio playback begins as soon as synthesis is available.

Perceived latency is often introduced outside the speech models themselves.

Agent Fails to Respond / Dead Air

Usually originates in:
Turn detection failure, orchestration bugs, or downstream service timeout.

What to inspect:
Whether EndOfTurn events are being delivered, LLM response timeouts, synthesis initialization, and event handler errors. Check for unhandled exceptions or blocking operations that prevent the agent from entering the response phase.

Inaccurate Transcription of Domain-Specific Language

Usually originates in:
Model selection or vocabulary coverage.

What to inspect:
Whether the ASR model is appropriate for the domain and whether specialized terminology is being surfaced to the speech system. Generic models may struggle with clinical, financial, or product-specific language without adaptation.

Agent Talks Over the User or Misses Interruptions

Usually originates in:
Audio transport or interruption detection.

What to inspect:
Whether audio is flowing continuously during agent playback and whether the system supports true full-duplex streaming. Reliable barge-in depends on uninterrupted microphone input and fast speech-start detection.

Agent Responds Too Early / Premature Interruption

Usually originates in:
Aggressive end-of-turn thresholds or speech boundary tuning.

What to inspect:
EOT threshold configuration (eot_threshold, eager_eot_threshold), whether partial transcripts or early turn signals trigger responses prematurely, and VAD sensitivity settings. Systems optimized for speed may sacrifice accuracy in detecting natural pauses.

Choppy, Distorted, or Unnatural Audio

Usually originates in:
Playback buffering, encoding mismatches, or network instability.

What to inspect:
Consistency between audio formats across the pipeline and whether playback strategies introduce unnecessary buffering. Many perceived synthesis issues are actually transport or client-side artifacts.

Echo or Audio Feedback Loops

Usually originates in:
Audio routing configuration or lack of echo cancellation.

What to inspect:
Whether agent output is being fed back into the input stream, full-duplex configuration, and hardware echo cancellation settings. In telephony scenarios, verify that the media gateway properly isolates inbound and outbound audio channels. Agent hearing itself can trigger response loops or distorted transcription.

Tools or Function Calls Are Not Triggered

Usually originates in:
Tool visibility or instruction framing.

What to inspect:
Whether functions are clearly defined, discoverable to the reasoning model, and aligned with the agent's role. If tools are underspecified or poorly scoped, the model may default to answering directly rather than invoking them.

Repetitive or Incoherent Responses Over Time

Usually originates in:
Context growth or prompt design.

What to inspect:
How conversational history is accumulated, summarized, or truncated. Long-running sessions often degrade when older context overwhelms the model's working memory or is repeated unintentionally.

Authentication or Connection Failures

Usually originates in:
Token lifecycle management or endpoint configuration.

What to inspect:
Whether credentials are valid, properly scoped, and refreshed as needed, and whether the correct regional or feature-enabled endpoints are being used.

WebSocket Disconnections or Connection Instability

Usually originates in:
Network reliability, keepalive configuration, or session recovery logic.

What to inspect:
Connection timeout settings, reconnection backoff logic, whether state is properly restored after reconnect, and network proxy or firewall configurations. Long-lived streaming sessions require explicit keepalive mechanisms and graceful reconnection handling to maintain conversational continuity.

Missing or Confused Speakers in Multi-Party Scenarios

Usually originates in:
Audio routing or channel configuration.

What to inspect:
Whether speakers are mixed correctly upstream and whether the architecture supports the intended number of participants. Many real-time agents assume a single human speaker per session.

Loss of Context or "Forgetting" Mid-Conversation

Usually originates in:
State persistence or memory strategy.

What to inspect:
How prior turns are retained and injected into reasoning. Long or complex interactions often require explicit summarization or structured memory rather than raw transcript accumulation.

Closing Note

Most real-time voice issues fall into one of five layers: **audio capture, transcription, reasoning, synthesis, or playback**. Diagnosing problems effectively requires identifying **which layer is responsible before attempting to tune parameters or swap components**.

For implementation details, configuration options, and code-level fixes, refer to Deepgram's official documentation and reference implementations. This appendix is intended to help you reason about failures architecturally, not replace production debugging workflows.

Deepgram

deepgram.com