ecosmob
makes your vision a reality

The Ultimate

AI Integration

Checklist for FreeSWITCH Deployments

Abstract

Building an AI voice agent sounds deceptively simple: wire up FreeSWITCH, pipe the audio to an LLM, and let the magic happen.
But between the concept and the reality sits a graveyard of failed deployments: calls that feel sluggish, conversations that lag, and transcriptions that miss half the words. The difference between a bot that feels responsive and one that feels dead is measured in milliseconds.

This whitepaper walks through the full technical **blueprint for FreeSWITCH automation with AI, moving beyond generic approaches to a systematic architecture built for human-quality latency and reliability.**
We've learned from real production implementations. And here's what actually works.

Why Standard **FreeSWITCH AI Integrations Fail**

Most teams approaching AI + FreeSWITCH integration start with a simple assumption: throw the audio at an API, get text back, send it to an LLM, synthesize a response, and bridge it back.
In theory, the architecture is straightforward. In practice, this approach produces voice agents that users hang up on within the first exchange.

The problem isn't the concept.

The problem is that the industry standard blueprints don't account for the reality of voice interaction, which is that sub-second response latency is non-negotiable.
When a user hears more than 1.5 seconds of silence after speaking, they assume the system broke. When they hear more than 2 seconds, they've already decided the experience is poor.

Every layer in a voice **AI stack adds latency**

- Audio capture adds a few milliseconds.
- Transmission across the network adds 50–100ms.
- Speech-to-text (STT) processing can take 200–400ms.
- The LLM generating a response can take anywhere from 300ms to several seconds, depending on the model and infrastructure.
- Text-to-speech (TTS) synthesis adds another 100–300ms.
- Then the audio has to travel back to the user.

All told, a poorly tuned system can easily hit 3–5 seconds of response delay before the user ever hears a word from the agent.

Worse, standard VoIP architectures (with their PSTN-era assumptions about codec choice, jitter buffer sizing, and media handling) work against you.

- An 8 kHz codec optimized for narrowband PSTN calls will torpedo transcription accuracy when fed into a modern speech-to-text engine trained on 16 kHz or higher.
- Generic jitter buffers tuned for voice calls will add 40–60ms of unnecessary delay when you're trying to shave milliseconds.
- Simple SIP-based orchestration introduces round-trip delays that compound across the pipeline.

This whitepaper will give you the FreeSWITCH deployment checklist and architectural decisions required for FreeSWITCH automation with AI without falling into these traps.

***The goal:** conversational, sub-500 ms end-to-end latency that makes callers forget they're talking to a machine.*

Understanding the **Modern AI Voice Stack**

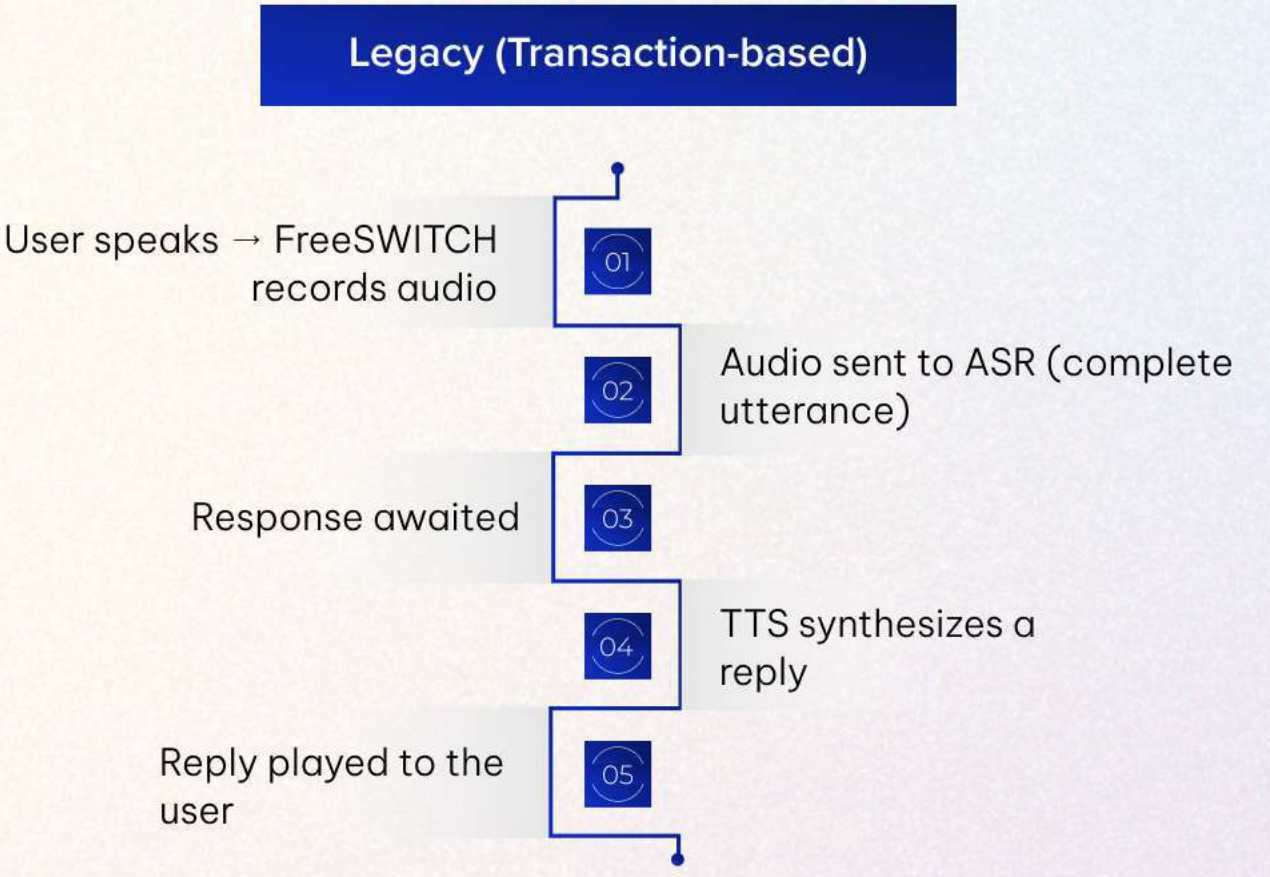
Before we dive into the checklist, it's essential to understand how AI voice differs fundamentally from traditional VoIP.

Legacy vs. Modern Architecture

For decades, FreeSWITCH was built to handle transactional IVR (record a call, play a prompt, collect a DTMF digit, route based on that choice). Media flow was simple, point-to-point, low-latency required, but latency itself wasn't mission-critical. A 500ms delay in answering a call didn't matter.

AI voice changes this entirely. A modern voice AI stack requires real-time, bidirectional streaming of media to an orchestrator. The orchestrator listens to the user while simultaneously deciding what to say next, and it needs to interrupt itself mid-sentence if the user starts talking (barge-in).

Here's the critical architectural shift:



Legacy (Transaction-based)

01 User speaks → Audio streamed in real-time via WebSocket

02 STT processes partial utterances (incremental results)

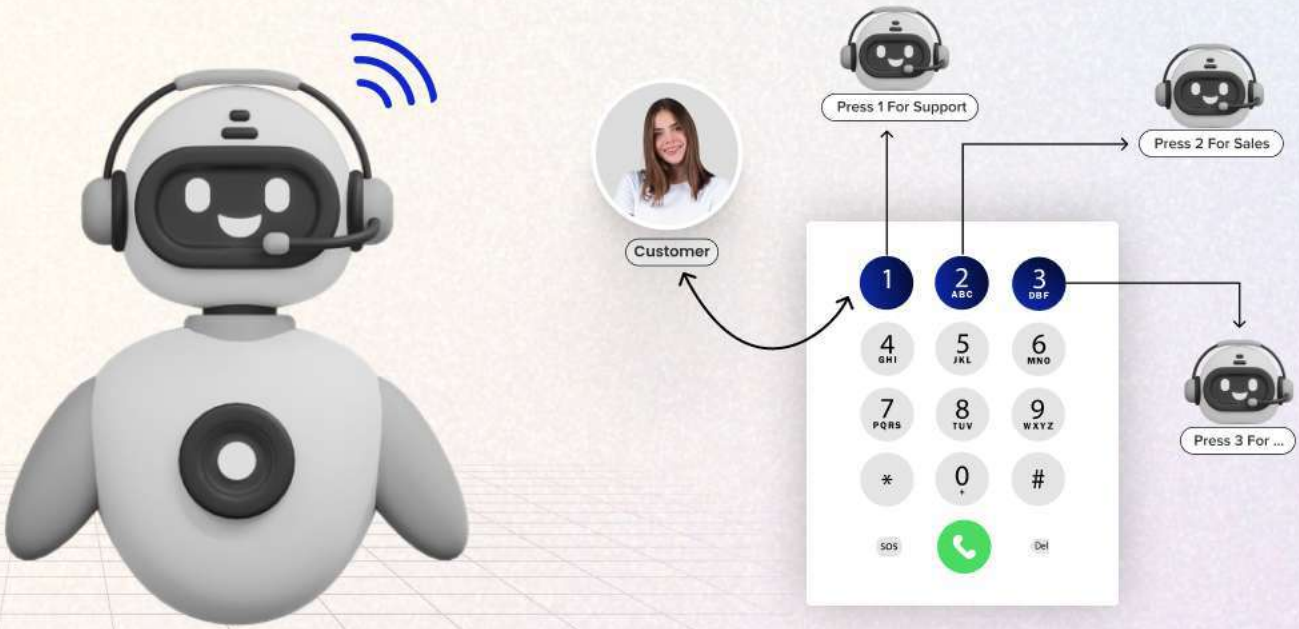
03 LLM begins formulating a response while the user is still speaking

04 TTS starts synthesizing the first sentence before the response is complete

05 If the user interrupts, the system detects speech energy and cuts off TTS immediately

06 Next response generation begins without delay

The difference is that traditional IVR waits for completion at each stage. Whereas modern AI systems pipeline every stage and make decisions incrementally.
This requires different infrastructure, different codec choices, different media handling, and different orchestration logic.



The Integration **Patterns**

There are two primary patterns for connecting AI to FreeSWITCH:

Pattern 1

MRCP (Legacy, Declining)

mod_unimrcp bridges FreeSWITCH to an MRCP server (Speech-to-Text and Text-to-Speech services). This was the standard for years and is still widely deployed.
However, MRCP is fundamentally request–response based, meaning it's built around discrete speech recognition and synthesis tasks, not continuous streaming. This makes it a poor fit for real-time voice AI that needs to interrupt, overlap, and stream.
If you're locked into MRCP due to legacy constraints, you can still optimize, but there's an inherent latency ceiling.

Pattern 2

Stream-Based (Modern, Recommended)

This is where real-time AI happens.
Media is streamed bidirectionally via WebSocket (or raw UDP/TCP) to an external orchestrator service. The orchestrator manages the complete AI pipeline (STT, LLM, TTS, barge-in logic) and streams responses back in real time. This is the architecture used by modern voice AI platforms.
It's also the path **Ecosmob's VoiceBot Connector** uses, abstracting the WebSocket orchestration layer so you don't have to hand-roll it.

For this whitepaper, we focus on Pattern 2, as it's the only architecture that reliably hits the sub-500ms latency targets required for a natural conversation.

The FreeSWITCH AI Integration Checklist Phase 1

(Infrastructure & Media Tuning)

Before you ever write a line of orchestration code, you need to ensure FreeSWITCH is configured for low-latency, high-quality audio. This layer is often overlooked, but it's where many deployments hemorrhage latency and quality.

Checklist Item 1.1

Codec Selection

Your codec choice is the first domino, and here's why it matters more than most realize:
When you configure FreeSWITCH with G.711 (PCMU/PCMA), you're using 8 kHz audio (the narrowband standard inherited from PSTN). For traditional voice calls, this is fine. For AI voice processing, it's a problem.

Modern speech-to-text engines (Deepgram, AssemblyAI, Google Cloud Speech-to-Text, OpenAI Whisper, and others) are trained on wideband audio (16 kHz or higher).
So when you send them 8 kHz audio, the (Automatic Speech Recognition) ASR engine treats it as lower-quality input. You'll see Word Error Rates (WER) spike. Over a conversation, that translates to missed intents, misunderstood requests, and user frustration.
Additionally, when you force the AI orchestrator to upsample from 8 kHz to 16 kHz internally, you're adding 10–20ms of unnecessary latency just for audio preprocessing.

The right choice

G.722 or Opus

G.722 is a 16 kHz wideband codec with excellent support in FreeSWITCH.
It compresses reasonably well (64 kbps typical) while giving you the sample rate modern AI models expect. Opus goes further by offering superior compression, adaptive bitrate, and quality up to 48 kHz. It's the default in WebRTC for a reason.

Configure FreeSWITCH like this:

```
<profile name="internal">
  <param name="inbound-codec-prefs" value="G722,Opus,PCMU"/>
  <param name="outbound-codec-prefs" value="G722,Opus,PCMU"/>
  ...
</profile>
```

Push G.722 and Opus to the front of the negotiation list. Some endpoints will still negotiate G.711 (keep it as a fallback), but for AI-connected legs, enforce wideband wherever possible.

Important note on codec transcoding

FreeSWITCH will automatically transcode between codecs if two legs negotiate different ones. This is convenient but adds CPU overhead and latency. If the incoming SIP provider forces G.711 and your AI orchestrator expects 16 kHz audio, FreeSWITCH will decode and re-encode the audio. For a handful of calls, that's fine; at scale, it's wasteful.
The ideal scenario: negotiate wideband codecs end-to-end and avoid transcoding on AI-critical paths.

Our Experts Suggest

Don't treat codec negotiation as a democracy. If you control the SIP profiles for your AI integration, don't "negotiate", mandate. On external legs (carriers, unknown devices), accept G.711. On AI legs, force G.722 or Opus and let FreeSWITCH transcode only at the boundary.
The marginal CPU cost is trivial compared to the improvement in ASR accuracy and the time you save debugging "mysterious" misrecognitions caused by narrowband audio.

Checklist Item 1.2

Jitter Buffer Tuning

This is where most teams leave latency on the table.
Out of the box, FreeSWITCH ships with a generic jitter buffer designed for voice calls on unstable networks. It's conservative, meaning it buffers extra audio to absorb packet jitter and loss. For traditional voice, that's a sensible trade-off. For AI, it's often a mistake.

A typical jitter buffer might hold 60–120ms of audio by default. On a stable network (which your AI orchestrator path should be), this is pure latency waste; you're sitting on packets instead of streaming them.

Here's the tuning strategy: For the AI orchestrator bridge, disable or minimize the jitter buffer. If you're bridging to an orchestrator on your own network or a reliable cloud region, you don't need much buffering.

Disable it entirely:

```
<action application="set" data="rtp_jitter_buffer_during_bridge=false"/>
```

Or constrain it aggressively:

```
<action application="set" data="jitterbuffer_msec=20:40"/>
```

This tells FreeSWITCH: use 20ms as your safety margin, cap at 40ms, and don't grow beyond that.

For external caller legs (inbound): the caller may be on mobile data, a poor Wi-Fi, or a noisy PSTN trunk.

In that case, use a moderate jitter buffer:

```
<action application="set" data="jitterbuffer_msec=60:100"/>
```

The asymmetric design is deliberate. There's conservative buffering on the inbound leg where the network is messy and aggressive. And low-latency behavior on the AI leg, where you control the environment.

Our Experts Suggest

Before touching jitter buffer settings, run a simple baseline between your FreeSWITCH host and the AI orchestrator endpoint. Use mtr or ping over a long interval (several minutes) to get realistic RTT and packet-loss data.
If you see <5ms RTT and 0% loss in the same data center, turn jitter buffers off for AI legs and keep them off. If you see 20–50ms RTT and negligible loss across regions, a fixed 20ms buffer is usually enough.
Adaptive jitter buffers that grow over time are a crutch for bad networks; on AI paths, they mostly just pad your latency budget.

Checklist Item 1.3

Kernel Timing & System Configuration

This is deeper in the stack, but it has a visible effect on real-time performance.
FreeSWITCH media timing relies on kernel timers to pace RTP transmission and process audio frames at precise intervals. On systems with poor timer resolution or high system load, RTP frames can arrive in small bursts instead of being evenly spaced. That jitter forces you to use larger jitter buffers, which again translates into latency.
Here are some key points to check:

1. High-Resolution Timers (HRT) in Linux

Modern Linux kernels support high-resolution timers **(timerfd)**. FreeSWITCH will use them when they're available.

Check:

```
cat /proc/sys/kernel/hrtimer_resolution
```

If you see something in nanoseconds, you're in good shape. If it's in microseconds, you're on older or non-optimized settings. On a serious production box, you want HRT enabled in the kernel configuration.

2. CPU Isolation for FreeSWITCH

If FreeSWITCH shares a node with other heavy workloads (databases, analytics, batch jobs), kernel context switching will show up as jitter. For serious deployments, pin FreeSWITCH's core processes to dedicated CPU cores and use CPU isolation so background processes don't steal cycles.

3. Interrupt Latency Monitoring

High interrupt latency (from NICs, storage, etc.) can cause FreeSWITCH to miss its media timing slots. Tools like cyclictest or latencytop help you see whether you're consistently hitting tight timing targets.
If you're regularly seeing >100µs latency on a supposed "real-time" box, you have tuning work to do (driver options, IRQ affinity, power-saving features, etc.).
Most voice deployments never touch this layer, but once you do, it's the difference between "usually fine" and "predictably excellent."

Our Experts Suggest

Treat your FreeSWITCH host like you'd treat a low-latency trading platform or a media mixing engine, not like a general-purpose web server. Turn off aggressive CPU power-saving modes, don't schedule heavy cron jobs during call peaks, and pin your heaviest FreeSWITCH threads to isolated cores. Then measure p50, p95, and p99 media processing latency over a full day. If your p99 is much worse than your p50, you're seeing rare but painful spikes—exactly the kind that make one in a hundred calls feel broken.

Checklist Item 1.4

SIP Profile Optimization

Your SIP profile configuration influences how media is negotiated and handled.

A representative internal profile:

```
<profile name="internal">
  <!-- Media codec prefs -->
  <param name="inbound-codec-prefs" value="G722,Opus,PCMU"/>
  <param name="outbound-codec-prefs" value="G722,Opus,PCMU"/>

  <!-- Disable VAD for AI legs (we'll handle it smartly at the orchestrator) -->
  <param name="vad" value="none"/>

  <!-- Disable CNG (comfort noise generation) on AI legs -->
  <!-- Many TTS engines and AI pipelines already model their own noise floor -->

  <!-- Enable candidate pruning to speed up SDP negotiation -->
  <param name="candidate-prune" value="aggressive"/>

  <!-- Disable some legacy SIP behavior that adds overhead -->
  <param name="username-to" value="false"/>
</profile>
```

The philosophy here is straightforward: disable legacy safety nets that made sense on PSTN-era networks but slow you down in controlled cloud environments.

The FreeSWITCH **AI Integration Checklist Phase 2**

The Integration Layer

With the media foundation in place, we can move up to the orchestration layer—the logic that actually turns raw audio into conversation.

Checklist Item 2.1

Stream Architecture (ESL vs WebSocket vs Media Bug)

You have three families of approaches to ferry audio from FreeSWITCH to your AI stack.

Option A: Event Socket Library (ESL) + External Recording

ESL gives you programmatic control: send commands, receive events, and manage calls.
Some teams combine ESL with recording to a FIFO or file, then feed that into STT.

Pros

- Mature, well supported
- Excellent for call control (transfer, routing, state)

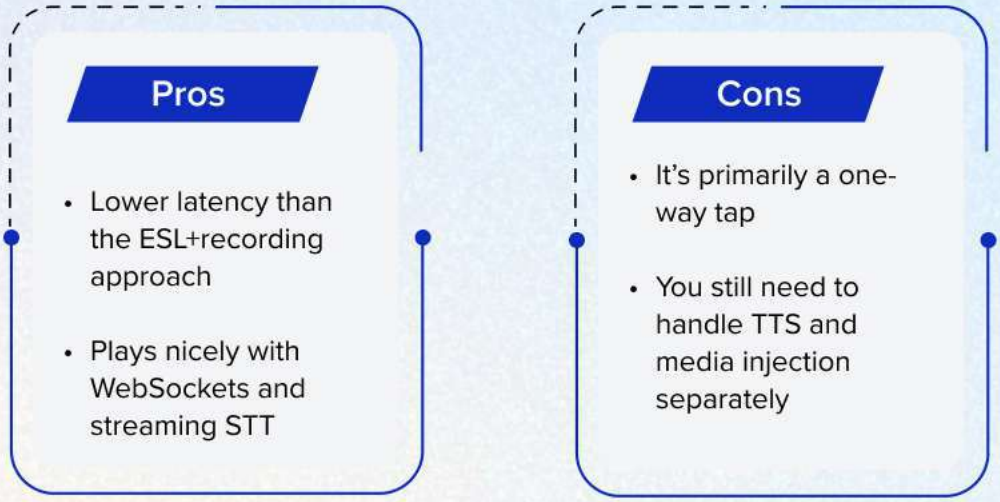
Cons

- Extra buffering when recording audio
- Latency from writing/reading from disk or pipes
- Not designed for continuous, bidirectional, low-latency streaming

You can make this work for basic “press 1 for balance” scenarios, but it struggles for natural, back-and-forth conversations.

Option B: mod_audio_fork

mod_audio_fork taps into the media path and forks audio to an external endpoint over TCP/WebSocket. This gets you much closer to "live" audio in your orchestrator.



It's a strong building block, but you still have work to do to support true bidirectional AI.

Option C: Full Stream-Based Bidirectional Orchestration (Recommended)

This is what modern stacks converge on. Your orchestrator service maintains a WebSocket connection where:

- Incoming audio from FreeSWITCH streams in real time
- STT, LLM, and TTS sit behind the orchestrator
- Outgoing synthesized audio streams back to FreeSWITCH and to the user

This is the pattern Ecosmob's VoiceBot Connector is designed to implement. It's also the only pattern that supports:

- Sub-second response consistently
- True barge-in (interrupting mid-utterance)
- Smooth handoff to live agents

The trade-off is complexity. But for real conversational AI, this is the arena you're in.

Our Experts Suggest

If you're going custom, pick one stack and do it well.

- **For Python, websockets + aiohttp + a small audio layer (e.g., soundfile/pydub) are enough to build a robust orchestrator.**
- **For Node.js, ws + a minimal PCM utility library works.**

Whichever you choose, keep that orchestrator in its own process or container; don't embed it inside FreeSWITCH or mix it with unrelated application logic. You want to be able to scale, restart, or roll back your AI layer independently without touching the media server.

Checklist Item 2.2

Voice Activity Detection (VAD) and Barge-In

This is where the "AI" either feels natural or robotic.
A human conversation is full of interruptions and overlaps. The ability for a user to cut in and redirect the conversation is non-negotiable.
Achieving this in a machine-driven system means:

1. Detecting when the user starts speaking
2. Cutting off TTS immediately
3. Feeding the new utterance back into the pipeline without delay

Naively, you might measure average signal energy and call anything above a threshold “speech.” That works acceptably in a clean lab, and poorly everywhere else.
Modern VAD solutions use compact neural models trained to distinguish speech from non-speech. These are baked into many STT engines and also offered as standalone services.

A practical FreeSWITCH + AI flow looks like:

1. User speech is streamed to the orchestrator.
   - The orchestrator runs a VAD model on the incoming stream.

2. On speech onset, the orchestrator emits a speech_started or equivalent event.
   - This is your cue that the user is trying to take the floor.

3. The orchestrator sends a “break” command back to FreeSWITCH.
   - In FreeSWITCH terms, this is typically a uuid_break to stop any ongoing playback on the channel.

4. The existing TTS stream is stopped, and the user's utterance is processed.

Dialplan example to give the orchestrator control:

```
<action application="bridge"
data="{hangup_after_bridge=false,execute_on_answer='socket
localhost:8085 async full'}sofia/gateway/ai_orchestrator/call"/>
```

In Node.js, the barge-in handler might look like:

```
// When VAD detects speech onset:
orchestrator.on('speech_started', async (callUuid) => {
  // Tell FreeSWITCH to stop playing any audio
  await eslConnection.api('uuid_break', callUuid, 'all');
  // Stop the current TTS stream in your orchestrator
  orchestrator.stopTtsStream(callUuid);
  // Resume STT on the new segment and process it
  orchestrator.restartSttProcessing(callUuid);
});
```

Tuning VAD

You'll typically get access to parameters such as:

- **threshold:** how confident the model must be that this is speech
- **silence_duration_ms :** how long silence must persist before you decide speech is over
- **prefix_padding_ms:** how many milliseconds of audio before the detected start you want to prepend (to avoid clipping the start of words)

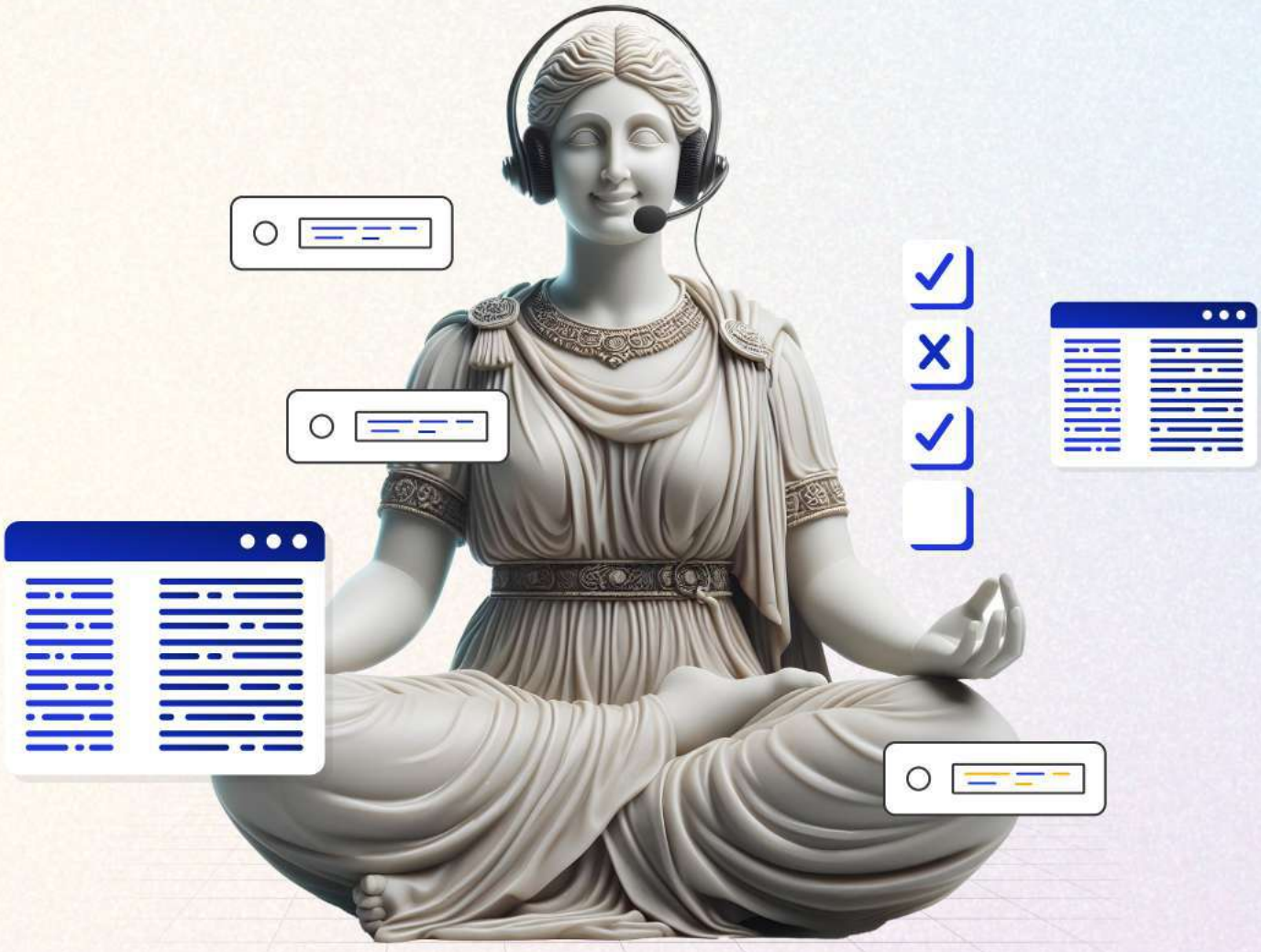
A sane starting point:

- threshold: 0.5
- silence_duration_ms: 500
- prefix_padding_ms: 100

Then you iterate with real calls. A slightly over-sensitive VAD that occasionally triggers early often feels better than a sluggish one that makes the user repeat themselves.

Our Experts Suggest

Don’t reinvent VAD. Use a battle-tested implementation; either the one bundled with your STT provider or a well-maintained open-source project.
Your job is to wire it into your orchestrator and tune it to your environment, not to design signal-processing algorithms from scratch. And when you tune, use real call recordings, not canned lab samples; office noise, headset echo, and mobile background sound will surface edge cases that synthetic tests never reveal.



Checklist Item 2.3

Latency Budgeting

Once the architecture is built, you need a clear latency budget (where you spend time, and what "good" looks like).

Here's a realistic voice AI budget:

Component	Typical Range	Good	Excellent
Network (user → FreeSWITCH)	10–100 ms	< 30 ms	< 15 ms
Client audio buffer	20–100 ms	< 50 ms	< 20 ms
FreeSWITCH media processing	10–30 ms	< 20 ms	< 10 ms
Network (FreeSWITCH → orchestrator)	5–50 ms	< 20 ms	< 10 ms
STT (streaming)	100–500 ms	< 200 ms	< 150 ms
LLM inference	200–2000 ms	< 500 ms	< 300 ms
TTS synthesis	100–500 ms	< 200 ms	< 150 ms
Playback buffer	5–50 ms	< 20 ms	< 10 ms
Network (orchestrator → FreeSWITCH)	20–100 ms	< 50 ms	< 20 ms
Total E2E	500–3500 ms	< 1000 ms	< 700 ms

Most deployments blow their budget in one place: LLM latency.

Here's how you can optimize that:

1. Use smaller models when the task allows it (routing, FAQ, triage). Save the heavy models for where they're actually necessary.
2. Cache aggressively when responses are repetitive (billing questions, common intents).
3. Use streaming responses and start TTS on the first tokens instead of waiting for the full reply.
4. Keep prompts lean (over-prompting an LLM wastes both latency and money).

Our Experts Suggest

Treat latency as a first-class Service Level Objective (SLO), not an afterthought.

- Build tracing into your orchestrator so every call logs STT, LLM, and TTS durations separately.
- Look at p90/p95 metrics weekly.
- If your p95 for LLM jumps from 400ms to 900ms after a deployment, you want to know immediately, not after customers complain that "the bot has become slow again."

The FreeSWITCH AI Integration Checklist Phase 3

Security, Resilience & Fallback

A system that only works on a good day isn't a production system.

Checklist Item 3.1

Handling Orchestrator Failures

Your AI orchestration layer will fail at some point with model timeouts, provider outages, bugs, network splits, etc. You need a plan for those calls. Let's look at the two common patterns.

Failover to a Human Agent

If the orchestrator doesn't respond within a strict timeout (for example, 5 seconds), transfer the call to a queue.

```
<action application="set" data="orchestrator_timeout=5000"/>
<action application="bridge" data="{originate_timeout=5}sofia/
gateway/ai_orchestrator/call"/>

<action application="set" data="call_state=orchestrator_failed"/>
<action application="transfer" data="9999 XML default"/>  <!--
Agent queue -->
```

Fallback to Rigid IVR

If human escalation isn't available, drop to a simple DTMF menu instead of leaving the caller in limbo:

```
<action application="set" data="call_state=orchestrator_failed"/>
<action application="playback" data="say:The AI system is
temporarily unavailable. Press 1 for billing, Press 2 for support..."/>
<action application="play_and_detect_speech" data="..."/>
```

***The principle:* no silent failure states. Every orchestrator failure should have an explicit call path.**

Checklist Item 3.2

PII Detection and Redaction

Live callers will say sensitive things like card numbers, social security numbers, account IDs, and medical details.

You need guardrails at three points:

1. Before sending audio or text to external providers.
2. Before storing logs or transcripts.
3. Before exposing transcripts in tools or dashboards.

A pragmatic workflow:

- Run a PII detection step in parallel with STT results (using something like Microsoft Presidio or a similar library).
- Replace detected PII in text with placeholders ([SSN], [CARD]).
- For ultra-sensitive use cases, locally process those segments or route to a human.

You shouldn't be shipping raw card numbers to third-party APIs "by accident."

Our Experts Suggest

Don't bolt PII handling on at the end. Design your data model so that transcripts, recordings, and analytics all flow through the same redaction filter before they touch long-term storage. Use separate storage for any data that absolutely must remain in raw form (and keep that footprint as small and short-lived as possible).

Checklist Item 3.3

Call Recording and Compliance

Recording voice AI interactions raises compliance questions (GDPR, PCI, HIPAA, local telecom regulations).

This checklist will help avoid compliance troubles:

- Announce recording and obtain consent where required.
- Encrypt recordings at rest and in transit.
- Implement retention policies and automatic deletion after N days.
- Keep an audit trail of who accessed which recordings and when.

FreeSWITCH gives you primitives; your responsibility is how you wire them into your policies.

Checklist Item 3.4

Preventing Abuse

AI voice endpoints are attractive to attackers:

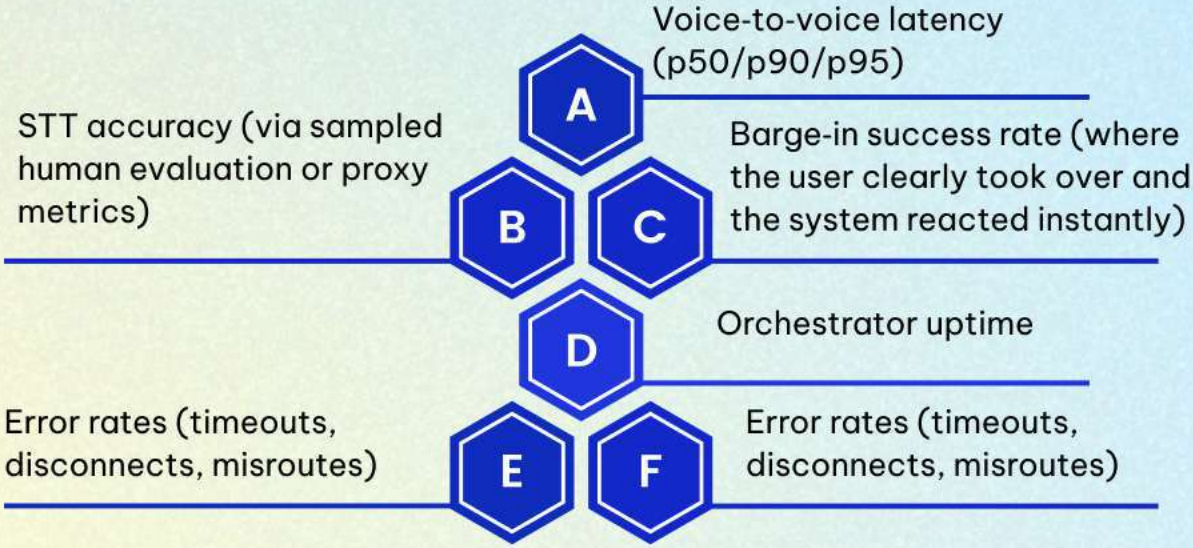
- Spammers hammering your entry points
- Replay attacks using recorded audio
- Prompt injection attempts through voice ("ignore all previous instructions...")

These mitigations will help:

- Rate-limit by IP and caller ID.
- Track abnormal patterns (thousands of short calls, repeated failures).
- Sanitize user text when it's converted from speech before injecting it into prompts.
- Consider simple audio fingerprinting to detect repeated audio segments.

Operational Excellence
(Monitoring, Metrics, and KPIs)

Key Metrics to Take Care Of



A simple structured log per call might include:

```
{
  "call_id": "abc123",
  "duration_seconds": 45,
  "latency_ms": {
    "voice_to_voice_total": 650,
    "stt": 180,
    "llm": 320,
    "tts": 150
  },
  "barge_in_count": 2,
  "transfers": 0,
  "disposition": "completed"
}
```

Alerting

You want to know about problems before your customers do.

Define and test these alerts:

- p95 E2E latency > 1200ms for 5 minutes → page
- Orchestrator error rate > 1% over 10 minutes → investigate
- Drop in the successful completion rate or spike in transfers to humans

Integration with **Ecosmob's VoiceBot Connector**

If you're building your own orchestrator, the checklist above is your roadmap.
If you'd like to accelerate the process and up the efficiency, Ecosmob built a VoiceBot Connector to solve the same problem with fewer moving parts on your side.

The Voicebot Connector:

- Bridges FreeSWITCH to your STT/LLM/TTS stack over WebSocket
- Handles barge-in and VAD logic
- Manages call state and handoff to human agents
- Implements the latency and resilience patterns described earlier

We deploy it alongside your FreeSWITCH system, configure where it should send audio and where it should get responses from, and let it take over the orchestration work that would otherwise live in your custom code.

The aim isn't to lock you into a black box,
but to give you a pre-tuned implementation of best practices so you can spend more time on your use cases.

The Difference **Between Failed and Thriving Deployments**

The difference between a voice AI deployment that feels magical and one that feels broken is measured in milliseconds and decisions.
The job doesn't stop at picking "the best model."

It's mainly about treating the stack end-to-end:

- High-quality, wideband audio in the right codecs
- Minimal, intentional buffering
- Stream-oriented integration rather than bolted-on APIs
- A realistic latency budget and the instrumentation to enforce it
- Thoughtful failure modes, not wishful thinking

Most failed deployments don't fall apart because of one catastrophic mistake.
They stack small oversights until the experience becomes sluggish and brittle.

The FreeSWITCH deployment checklist in this whitepaper is the antidote.
Whether you build everything yourself or lean on a connector like Ecosmob's Voicebot Connector, these are the principles that separate demo-level AI from production-grade voice automation.

ecosmob
makes your vision a reality

Reach US @

INDIA

501-503, Binori B Square 1, Nr.
Neptune House,Ambli -Bopal Road,
Ahmedabad-380058, Gujarat, India.

USA

300 SE 2nd Street, Suite 600 Ft.
Lauderdale,Florida, USA 33301

CANADA

1285 West Broadway, Suite 600,
Vancouver, BC, V6H 3X8

PHONE

+91 99988-51106 +1-303-997-3139 +1 (604) 900-8870

MAIL US AT

sales@ecosmob.com

Visit US