Building Open-Source Voice Bots: ASR Technology Overview

By ClearlyIP Published February 25, 2025 55 min read

Building a State-of-the-Art Open-Source Voice Bot: A Comprehensive Guide

1. Automatic Speech Recognition (ASR)

Open-source ASR technology forms the first stage of a voice bot pipeline, converting spoken audio into text. Modern ASR models are typically neural networks trained on vast speech datasets. **OpenAI Whisper** is a recent transformer-based model trained on 680k+ hours of multilingual data, achieving state-of-the-art accuracy on many benchmarks (Source: deepgram.com)(Source:

deepgram.com). Whisper comes in various sizes (Tiny to Large) balancing accuracy and speed; larger models yield lower word error rates (WER) but require more compute (GPUs for real-time use). **Kaldi**, by contrast, is a classic open-source toolkit using a modular HMM/DNN pipeline; it's highly customizable and supports training your own models, but its older architecture struggles to match modern end-to-end systems on open-domain audio (Source: deepgram.com). **Vosk** (from Alpha Cephei) builds on Kaldi with ready-to-use offline models in many languages (English, Chinese, Spanish, etc.), ranging from lightweight 40MB models up to 3GB for higher accuracy (Source: medium.com). Vosk is optimized for edge devices (even mobile or Raspberry Pi) and provides simple Python/C++ APIs for real-time transcription. Other notable open ASR projects include Facebook's **wav2vec 2.0** (self-supervised transformer models) and NVIDIA's **NeMo** QuartzNet models. In a comparative study, transformer models like Whisper significantly outperformed Kaldi on real-world speech: *"the Kaldi model produces pathologically bad WERs… the conventional pipeline model simply cannot compete"* in long-form audio tests (Source: deepgram.com). A robust wav2vec 2.0 model achieved better accuracy than Kaldi but *"significantly worse [WER] than Whisper across all domains"* (Source: deepgram.com). Simpler architectures (e.g. QuartzNet in NeMo) can offer fast inference; one benchmark found NeMo's QuartzNet had the fastest transcription time and high accuracy on clean audio, whereas Vosk was *"less accurate and slower"* than other neural models (Source: medium.com).

Training and Customization: Open-source ASR toolkits enable training or fine-tuning models on custom data. Kaldi provides recipes for training acoustic and language models from scratch given transcribed audio corpora. Mozilla's **DeepSpeech** (an older end-to-end model) similarly *"comes with a few pre-trained models and allows you to train your own"* (Source: rasa.com). Modern end-to-end models (Whisper, wav2vec, SpeechBrain) can be fine-tuned via transfer learning – e.g., a pre-trained model is adapted on domain-specific audio to improve accuracy on that domain (Source: jsaer.com). This is crucial if your bot's use-case involves specialized vocabulary or accents. Whisper's code is open-source, and while OpenAI didn't release a training script, the community has developed fine-tuning approaches. NVIDIA NeMo and Facebook's Fairseq provide scripts to fine-tune wav2vec or QuartzNet models on custom data. Vosk/Kaldi models can be adapted by training new n-gram language models or using Kaldi's chain models with your data. These open frameworks thus offer flexibility: you can use pre-trained models out of the box or invest in training custom models for higher accuracy in your domain (Source: jsaer.com).

Deployment: ASR models can be deployed locally or as services. Whisper and Vosk have Python APIs to transcribe audio files or microphone input in real-time. For scalable deployments, they can run in a server process (for example, Vosk provides a socket server mode and even Docker images per language). Kaldi can be integrated as a C++ library or served via REST in frameworks like KALDI

GStreamer server. When deploying ASR, consider real-time processing needs: streaming ASR (outputting partial results as speech is received) can greatly reduce latency for long utterances. Toolkits like Kaldi and Vosk support streaming decoding with voice activity detection. Whisper as released processes audio in blocks (not truly streaming), though smaller Whisper models (Tiny/Small) are fast enough for near-real-time transcription on GPUs. **Resource requirements:** Accuracy improvements often come at the cost of higher computation. For instance, Whisper's largest model (1.5B parameters) may add hundreds of milliseconds of latency (Source: developer.nvidia.com), so for real-time bots a medium or small model on GPU is a common trade-off. In summary, developers should choose an ASR engine that balances accuracy, language needs, and runtime performance – e.g. Whisper for best accuracy across many languages, Vosk/Kaldi for lightweight offline use, or wav2vec/QuartzNet models for a middle ground. It's also critical to consider preparing your LAN for a VoIP Phone System Deployment to ensure optimal performance for real-time voice applications.

2. Natural Language Understanding (NLU)

Once user speech is transcribed to text, the NLU component interprets the text to understand the user's intent and extract any key information. This step typically involves **Intent Recognition** (classifying which action or query the user wants) and **Entity Extraction** (identifying words/phrases representing key details like names, dates, amounts, etc.) (Source: botfriends.de). Open-source NLU frameworks greatly simplify this stage. **Rasa Open Source** (specifically the Rasa NLU module) is a popular choice that provides intent classification and entity extraction out-of-the-box. Rasa NLU uses a configurable *pipeline* of processors to turn user messages into intents and entities (Source: rasa.com). Developers can plug in different components – for example, a spaCy model for tokenization & named entity recognition, or a BERT-based classifier for intent – to tailor the NLU pipeline to their data. In fact, *"the structure of the Rasa NLU is fully configurable and defined with the help of the so-called pipeline"*, which can use either a **spaCy** backend (leveraging pretrained language models) or a **TensorFlow** backend (training neural network classifiers) (Source: botfriends.de). Rasa NLU supports multi-intent detection and provides prebuilt entities (e.g. common datetime or numeric extractors) while also allowing custom entity definitions trained on examples (Source: botfriends.de).

Other open-source NLP libraries complement or can even replace Rasa's NLU in certain cases. **spaCy**, a general-purpose NLP library, offers industrial-strength named entity recognition models for many languages, part-of-speech tagging, and efficient text processing. spaCy can be used within Rasa (via the SpacyTokenizer and SpacyEntityExtractor components) or standalone – for

example, one could use spaCy to detect entities in user text (like addresses, organizations, etc.) with its pre-trained models (Source: botfriends.de). However, spaCy doesn't natively do intent classification; you would pair it with a classifier (it does have a text categorizer component or one could use scikit-learn or **Haystack** etc.). **Haystack** (by deepset) is another open-source framework relevant to NLU, particularly for question-answering and retrieval-based understanding. Haystack provides modular pipelines to perform **retrieval augmented QA**, semantic search, and even conversational question answering over documents (Source: github.com). In a voice bot context, Haystack could be integrated to handle queries that require searching a knowledge base. For example, if a user asks a factual question, the bot's DM could invoke a Haystack pipeline that retrieves relevant documents and finds an answer span. This essentially extends NLU to not just classify intent but *understand and answer* open-domain questions using retrieved knowledge – a powerful capability using only open-source models (Haystack can leverage Transformers like DPR for retrieval and BERT for extractive QA).

Beyond these, developers can leverage Hugging Face Transformers to build custom NLU components. Fine-tuning a language model (like BERT, DistilBERT, or even GPT-2) on an intent classification dataset often yields high accuracy for intent recognition. There are also open-source pre-trained models for intent detection and slot filling (for instance, on the SNIPS or ATIS datasets) that can be adapted. Libraries like **DeepPavlov** (an open-source conversational AI library) offer ready-made NLU modules as well – e.g., intent classifiers and entity extractors for Russian and English, which could be useful in certain languages. **Snips NLU** (an older open-source library from the Snips voice assistant project) is another example; it focused on offline intent/entity parsing with lightweight models. While Snips is no longer maintained, its ideas (embedding-based intent classification and regex/dictionary-based entities) influenced many modern systems.

In summary, open-source NLU provides the tools to parse user utterances into structured data. A typical configuration might use Rasa NLU's machine learning pipeline to recognize intents (with an algorithm like the DIET classifier or a Transformer) and extract entities (using built-in CRF or spaCy), possibly supplemented by specialized components (e.g., Duckling for dates and numbers, which Rasa can integrate). With these tools, the voice bot can convert the raw text "Book a flight from NYC to London tomorrow" into a machine-readable intent (`BookFlight`) with entities like `origin: NYC`, `destination: London`, `date: tomorrow`. This structured output then feeds into the dialogue manager for the next step.

3. Dialogue Management (DM)

Dialogue Management is the "brain" of the voice bot, responsible for maintaining conversational state, deciding how to respond, and orchestrating the conversation flow. Unlike single-turn NLU, DM handles **context** and multi-turn logic: it must track what the user has said previously, what the bot has done, and what needs to happen next to fulfill the user's goal. Open-source frameworks provide two main paradigms for DM: **rule-based/state machine approaches** and **ML-based policy learning**.

Rasa Core (the dialogue component of Rasa) is an example of a machine-learning policy approach. It maintains a dialogue state (a tracker store with the conversation's slot values, intent history, etc.) and uses policies (which can be learned from example dialogues or hand-crafted rules) to choose the next action. Rasa's core policies include a TED Policy (Transformer Embedding Dialogue) that learns to predict the next bot action based on the conversation history, and rule-based policies for handling things like FAQs or forms. *"Rasa… consists of Rasa NLU (for intent recognition and entity extraction) and Rasa Core (for managing conversations)", using "machine learning-based policies for dialogue management."* (Source: keencomputer.com). This means that given training stories (example conversations), Rasa can generalize to unseen dialogues and decide, for instance, to prompt for missing info or handle clarification questions. Rasa's DM is highly configurable – developers can impose rules (e.g. always ask for confirmation if a price is above X) alongside the ML policies, achieving a blend of consistency and learning. Importantly, Rasa Core keeps track of **slots** (key pieces of info filled during the dialogue) and **conversation intents** to maintain context (for example, remembering the user's name mentioned earlier to use later, or carrying over an implicit context like the current topic). This enables handling context-dependent queries ("next week" meaning next week's schedule given the earlier context of scheduling, etc.).

Botpress is a prominent open-source platform that exemplifies the state-machine approach with a visual builder. Botpress is built on Node.js and uses a modular plugin architecture (Source: keencomputer.com). Conversations in Botpress are designed as flows or graphs: you define nodes (questions, messages) and transitions based on user intents or entities. Essentially, *Botpress "uses a state machine for managing conversation flows"* (Source: keencomputer.com). This approach is very intuitive for well-defined tasks – it's like drawing a flowchart of the dialogue. Botpress provides a GUI where non-programmers can design conversation paths, which is great for straightforward FAQ bots or form-filling dialogues. The trade-off is that purely flow-based bots can be rigid; handling unexpected user input requires defining many transitions or fallback handlers. That's where Botpress can integrate with NLU (it can use its built-in NLU or connect to Rasa, Dialogflow,

etc.) to trigger different flows. Overall, Botpress is ideal if you want a **low-code** solution with quick setup, while Rasa is better suited for ML-heavy, complex dialogues requiring learning and flexibility (Source: keencomputer.com).

Other open DM frameworks include **Microsoft Bot Framework SDK** (open-source libraries for building dialog logic in C#/Python, albeit often used with Microsoft's LUIs or QnA Maker for NLU), **Dialogflow CX** (Google's, not open-source though), and academic projects like **OpenDial** or **DeepPavlov Conversational Framework**. There is also interest in end-to-end dialogue models (where a single neural model decides the response given conversation history, often used in open-domain chatbots). However, for a state-of-the-art voice assistant on open tech, a pipeline approach (explicit NLU -> DM -> NLG) is preferred for control and transparency.

In managing dialogue, key tasks include **context tracking** (remembering what the user said before; e.g., in slot-filling, which slots are already filled), **decision logic** (maybe the bot should branch to a sub-dialog to clarify something), and **integration** with backend services. For example, if a user asks "What's my bank balance?", the DM must recognize the intent, ensure security (maybe the user is authenticated), then call an API to retrieve the balance, and finally decide to respond with that info. Open-source DM frameworks allow custom code for such integrations: Rasa has the concept of *actions* (Python code that can be triggered to perform operations like database queries) and Botpress allows hooking into actions or APIs at certain nodes. Managing conversational state often means maintaining a **dialogue memory** – this could be as simple as a dictionary of slots (in task-oriented bots) or more complex like a dialogue state vector. Some frameworks separate a **Dialog State Tracking (DST)** component (especially in research literature), but in practice Rasa/Botpress incorporate state tracking internally (Rasa's tracker, Botpress context variables).

Context handling also means the bot should handle references and follow-ups. For instance, in a multi-turn exchange: *User:* "Find Italian restaurants in downtown." *Bot:* "Sure, for what date?" *User:* "Tomorrow at 7." – the second user utterance doesn't repeat the context ("restaurants in downtown") explicitly, but the DM must remember it. Rasa would have kept that info in a slot (e.g., `cuisine=Italian`, `location=downtown`) and not ask again, proceeding to make a reservation perhaps. Good DM design uses context to make interactions smooth – perhaps using context carryover and disambiguation if needed.

In summary, open-source DM solutions like Rasa Core and Botpress enable robust conversational logic: Rasa uses ML policies and supports complex contextual assistants trained on real dialogues (Source: keencomputer.com), whereas Botpress favors a deterministic flow approach with a user-friendly interface (Source: keencomputer.com). Both can manage state, call external services, and handle multi-turn interactions. A state-of-the-art voice bot might even combine approaches – e.g.,

using Rasa's ML for general conversation, but with some hard-coded rules for critical business logic (or using Botpress for certain guided flows embedded in a larger Rasa assistant). The result is a dialogue manager that can flexibly navigate conversations, keep track of context, and decide the bot's actions to achieve successful outcomes.

4. Natural Language Generation (NLG)

Natural Language Generation is the component that produces the bot's reply in fluent, natural language. After the Dialogue Manager decides what the bot should do or say next (for example, it might decide "inform the user of their account balance"), the NLG module formulates the actual textual response to be spoken. There are two main approaches to NLG in voice bots: **template-based (rule-based) generation** and **generative model-based** generation.

Template-Based NLG: This is a straightforward method where responses are constructed from pre-defined templates with slot placeholders. For example, a template for a balance query might be `"Your current balance is $[balance]."` – the bot will fill in the `[balance]` with the actual number. Rule-based generation can also include conditional logic (e.g., slight grammatical variations if a value is plural vs singular). This approach is reliable and ensures the responses are precise and on-brand, which is why many task-oriented assistants still use templated responses for critical information. Rasa, for instance, encourages defining **response templates** in the domain file (or response selector); each intent/action can be mapped to one or multiple text templates. The advantages are clarity and control: you won't get a weird or incorrect phrasing because you wrote the template. The downside is limited variety and inability to handle truly free-form conversations. Nonetheless, for many domains (support bots, transactional bots), templated NLG suffices. You can author a variety of phrases for each intent to introduce some randomness, but it's still a closed set of responses.

Generative NLG (Neural Models): With advances in NLP, it's possible to have a neural network generate responses in an open-ended way. Large Language Models (LLMs) and sequence-to-sequence models can take a dialogue context and generate a continuation. Open-source examples include **DialoGPT** (a chatbot model from Microsoft trained on 147M Reddit conversations) which is *"a GPT-2 model… powerful in open-domain dialogue systems."* (Source: huggingface.co). Another is Facebook's **BlenderBot** (e.g., BlenderBot 2.0 with long-term memory) which was open-sourced and can carry on extended conversations. For smaller-scale local deployments, one might use a distilled version of such models (for example, a 90M parameter DialoGPT or a 400M BlenderBot) to generate varied responses. The benefit of generative NLG is the flexibility – the bot can handle inputs that

weren't anticipated by templates, and can produce more conversational, contextually rich replies. This is particularly useful in social-chat or open-domain scenarios (like an assistant that can chit-chat or answer off-the-cuff questions). However, generative NLG comes with challenges: it may produce incorrect or irrelevant outputs if not carefully constrained, and it requires significant computational resources. In a voice bot, using a large generative model might introduce latency (a few hundred milliseconds or more to generate a response) which can affect real-time performance.

A practical strategy is often a **hybrid**: use templated or retrieval-based responses for known intents (ensuring accuracy for task-specific content), and perhaps use a generative model fallback for unhandled queries or casual conversation. There are open-source projects that enable this – for instance, **Rasa** allows integration of custom action code, so one could call an open-source model like GPT-J or an **OpenAI GPT-2** model (running locally via HuggingFace) to generate a response when no rule matches an intent. Another approach is **retrieval-based NLG** where the system selects the best response from a repository of candidate responses (this can be powered by embedding similarity; it's like an FAQ matcher but for full sentences).

When using open-source generative models, ensure you have appropriate **filters** or constraints to avoid unwanted outputs, since these models can sometimes produce inappropriate or hallucinated content. Fine-tuning them on your domain dialogs (if data is available) can also improve relevancy and safety. An example open project is **ParlAI** by Facebook, which contains recipes for training dialogue models and even some pre-trained conversational models that you can run.

In summary, NLG in a state-of-the-art voice bot can range from simple and safe (templates) to sophisticated and flexible (neural generation). For many enterprise voice assistants, **consistency and reliability** are paramount, so a controlled NLG (templated with slight variations) is used for transactional dialogues. In more open-ended AI assistants (think of Alexa's casual mode or a customer engagement bot), incorporating a generative model can make the bot feel more natural and less repetitive. All of this can be achieved with open-source tools: you might use Rasa's Response Selector or template system for the bulk of replies, and leverage open-source Transformer models (DialoGPT, etc.) for small talk or unforeseen queries. Just remember to evaluate the outputs carefully – the goal is a coherent, contextually appropriate response that the TTS can then speak out.

5. Text-to-Speech (TTS)

Text-to-Speech is the final stage of the voice bot pipeline, converting the text response into audible speech for the user. The landscape of open-source TTS has advanced significantly in recent years, with neural network-based TTS models achieving natural, human-like speech quality. Key factors to consider in TTS are voice quality (naturalness and clarity), latency (how quickly speech can be generated), and multilingual/support for custom voices.

Coqui TTS (and Mozilla TTS): Coqui TTS is a leading open-source TTS toolkit, which originated from Mozilla's TTS project. It provides a framework to train and run state-of-the-art TTS models (like Tacotron 2, FastSpeech2, VITS, etc.) and comes with many pre-trained voices in different languages. Coqui TTS supports voice cloning (training a new voice given a small sample, though quality varies) and multi-speaker models. Using Coqui, one can generate fairly natural speech; for best quality it supports *VITS models*, which produce very realistic prosody (Source: tderflinger.com). Coqui is a powerful toolkit, but note that as of late 2024, it faced uncertainty in commercial backing (the original Mozilla project ended, and Coqui as a company had to rely on community support) (Source: tderflinger.com). Nevertheless, the open-source community continues to maintain it. The latency of Coqui models depends on size – some smaller models can run real-time on CPU, while bigger ones may need GPU for faster-than-real-time synthesis.

Piper: Piper is an emerging open-source TTS engine focused on efficiency. It's developed in the context of the Rhasspy voice assistant project. Piper uses lightweight neural models (based on efficient architectures) and is optimized for speed and low resource usage. According to one review, *"from the four tested open source text-to-speech applications, my favorite is clearly Piper. It has the most natural sounding speech."* (Source: tderflinger.com). Piper models are small and can run on devices like Raspberry Pi 4, making them ideal for edge deployments. Despite its efficiency, Piper's audio quality is impressively natural (it uses techniques like quantization and optimized vocoders). It may not have as many languages or voices as Coqui, but it covers major ones and the community is growing. For example, Piper offers English voices that are nearly human-like in intonation and timbre, all with inference speeds that can easily achieve low latency on CPUs.

Mycroft Mimic 3: Mimic 3 is another open-source neural TTS, created for the Mycroft AI assistant. It supports SSML, multiple voices, and an interactive mode. However, as an open project it saw limited updates (last major commits were a few years ago) and it's under an AGPL license (Source: tderflinger.com). Quality-wise, Mimic3 voices are decent and it was designed to be run on-device for privacy (it can run on a Raspberry Pi for example). Given its maintenance status, developers might prefer Coqui or Piper unless they specifically need something from Mimic3.

Older / Other TTS: There are legacy TTS engines like **eSpeak NG** and **Festival** which are fully open-source. eSpeak NG covers 100+ languages, is extremely lightweight (written in C), and can run on tiny devices (Source: tderflinger.com). But it produces robotic-sounding speech that lacks the natural prosody of neural TTS. Such engines might be considered if you need ultra-fast, resource-minimal TTS or support for a less common language not covered by neural models (since eSpeak can at least pronounce many languages albeit mechanically). **Festival**/Flite are similar category (unit-selection or formant synthesis based), better than nothing but not human-like. Another noteworthy project is **Tortoise-TTS** – an open-source high-end TTS that can mimic voices with remarkable quality, but it's very slow (not suitable for real-time use, more for offline generation of longer speeches where quality is paramount).

Voice Quality and Latency: Neural TTS models are evaluated by metrics like Mean Opinion Score (MOS) from human listeners. Many open TTS systems can achieve MOS in the 4.0+ range (out of 5) which indicates high naturalness, close to human speech (Source: zilliz.com). Quality depends on training data (hours of recorded speech per voice) and model architecture. Models like VITS (which Coqui uses) generate speech end-to-end and can capture subtle prosodic elements. On the latency front, there are models designed for speed: FastSpeech2 and Multi-band MelGAN or HiFiGAN vocoders can generate speech much faster than real time. For example, generating one second of audio might take only 0.1s of processing on a modern CPU for a small model. When deploying a voice bot, one must ensure the TTS is fast enough so that there isn't a long gap after the bot decides on a response. Generally, a well-optimized TTS can produce audio on-the-fly, so the user perceives almost immediate response. If using larger models, running on a GPU or doing some optimizations (quantizing the model to 8-bit, etc.) can help. Open-source tools allow these tweaks: e.g., converting a PyTorch model to ONNX and using accelerated inference engines.

Multilingual and Custom Voices: Many open TTS solutions support multiple languages, but you often need specific models for each language. Coqui's model zoo, for instance, has community-contributed models for Spanish, German, Chinese, and more. If your voice bot needs to speak multiple languages, you might load different models or a multi-lingual model (some TTS models are trained on multiple languages, but quality might vary per language). Custom voice creation (to have a unique persona for your assistant) is possible with open source: you can train a model on a new voice if you have recordings. Projects like **YourTTS** (from Coqui) show zero-shot or low-shot voice cloning – giving the model a sample of a new speaker and synthesizing that voice. While open methods for voice cloning exist, the quality might not equal proprietary services like Google's or Amazon's, but it's improving. Keep in mind, creating a high-quality voice can require a substantial dataset (hours of clean recordings) and training time.

In conclusion, open-source TTS has you covered from high-quality neural voices (Coqui TTS, Piper) to ultra-lightweight engines (eSpeak). A state-of-the-art open voice bot would likely use a neural TTS for natural interaction. For example, you might run Coqui TTS with an English VITS voice for a near-human voice, achieving a pleasant user experience. If deploying on an embedded device, Piper or a smaller model could be used to ensure quick response. It's also wise to have a backup or error-handling – e.g., if the TTS fails or is too slow at the moment, an emergency fallback could be a pre-recorded phrase or a simpler TTS engine so the bot always responds. Thankfully, with open tools, you have full control over the voice output and can tune the system (pick a voice with the right gender/tone, adjust speaking rate, add SSML for effects, etc.) to create the desired persona for your voice assistant.

6. Pipeline Architecture: Integrating ASR, NLU, DM, NLG, and TTS

Building a complete voice bot requires **connecting all the components** – from the user's microphone input to the speaker output. The architecture typically follows a sequential pipeline: audio input -> ASR -> NLU -> Dialogue Manager -> NLG -> TTS -> audio output. In a real system, these components can be separate services or libraries within one application. The integration must handle real-time streaming and maintain the conversation state across turns.

! https://developer.nvidia.com/blog/creating-voice-based-virtual-assistants-using-nvidia-riva-and-rasa/

Figure: High-level architecture of a voice assistant pipeline (example from an open-source stack combining Rasa and Nvidia Riva). The user's voice input (left) is sent to an ASR service which returns a transcript. The text is then passed to the NLU/DM (in this case, Rasa's components) which determine the appropriate response. The response text is finally sent to a TTS service to generate audio, which is played back to the user. (Source: developer.nvidia.com)

In a typical interaction, the following steps occur: **(1)** The user presses a push-to-talk or says a wake word (like "Hey MyAssistant") to activate listening. The client application (which could be a mobile app, web app, or IoT device) captures the audio from the microphone. **(2)** The raw audio stream is sent to the ASR component. If using a streaming ASR, this happens in real-time over a socket or streaming API, and partial text results might be returned as the user speaks. If using a non-streaming ASR, the system might first perform VAD (voice activity detection) to determine when the user has finished speaking (silence detection), then send the complete audio for transcription. **(3)**

The ASR outputs text (the recognized transcription). The architecture then passes this text to the NLU. In a modular system, the client might send the text to an NLU service or library call. For instance, a "Rasa NLU server" could be running that accepts the text and returns an intent and entities JSON. In other setups, this NLU step is embedded in the dialogue manager process. **(4)** The Dialogue Manager receives the structured input (or raw text if it combines NLU internally) and updates the conversation state. It decides on the next action – which could be to fetch some info, ask a follow-up question, or formulate a final answer. This often involves policy rules, database/API calls, etc., as discussed earlier. **(5)** The DM produces a response in text form. This might be selecting a template or using NLG to generate the text. For example, the DM might know the intent is "WeatherQuery" and it fetched "22°C" as the result, then it chooses a response template: "The current temperature is 22 degrees." **(6)** This response text is handed to the TTS component for synthesis. The TTS generates an audio waveform (e.g., a WAV or byte stream). **(7)** Finally, that audio is played to the user through the client application's speaker. The user hears the bot's answer.

All these components need to work in concert, ideally with minimal delay to preserve a smooth conversational experience. One key consideration is **real-time vs batch processing**. For a voice assistant, real-time (or near real-time) processing is required so that the conversation feels natural. Each turn of the dialogue often should be processed in under a second (a common goal is the "< 300ms ASR + < 300ms NLU/DM + < 300ms TTS" kind of budget, since humans notice delays beyond ~0.2s (Source: developer.nvidia.com)). Achieving this might involve processing audio as it streams (so-called *streaming ASR* that can start producing text before the user finished talking). Some systems even do **barge-in** handling, where the assistant can start formulating a response before the user finishes (though this is advanced). In contrast, batch processing might be used for offline tasks like transcribing voicemails or analyzing recorded calls – there, latency isn't critical, and you might trade speed for higher accuracy by using a bigger model.

Integration Patterns: In designing the architecture, you can choose between a monolithic design or microservices. Monolithic would mean you have a single application that handles audio input, calls an ASR library, then processes NLU/DM, etc., internally. This is simpler and avoids network overhead between components, but harder to scale individual parts. The microservice approach treats each component as a separate service with a defined API. For example, you might have: an *ASR service* (accepts audio, returns text), an *NLU/DM service* (accepts text, returns response text – essentially a complete chatbot brain that outputs a reply given an input), and a *TTS service* (accepts text, returns audio). In the example architecture above, NVIDIA's Riva services were used for ASR and TTS, and Rasa handled NLU and DM (Source: developer.nvidia.com). A thin **connector or orchestrator** component (sometimes called a "middleware" or "voice gateway") is often used to route between them. In Rasa's voice interface demo, for instance, they implemented a connector that takes the

audio from the frontend, sends it to DeepSpeech ASR, then sends the text to Rasa and gets the response, then calls TTS and returns the audio to the frontend (Source: rasa.com)(Source: developer.nvidia.com). This connector could be implemented with synchronous calls or an asynchronous messaging system.

Scaling and Robustness: In a production setting, you may have many users interacting concurrently. The architecture should allow scaling the heavy components – ASR and TTS are typically the most CPU/GPU intensive. If using microservices, you could run multiple instances of the ASR service behind a load balancer to handle parallel audio streams. Similarly, multiple TTS workers can generate audio in parallel. Dialogue Management (NLU/DM) often can handle many sessions in one process (especially if it's just waiting on I/O or doing lightweight ML), but if using large NLU models, that too can be scaled out. Using a cloud-native approach (see Deployment section) with container orchestration (Kubernetes) can automate this scaling.

Real-Time Communication: The interfaces between components can be RESTful (HTTP APIs) or streaming protocols. For example, if the user is speaking for a long time, a streaming gRPC or WebSocket connection to the ASR service can continuously send partial transcripts, which could even allow the NLU/DM to start processing before the user finished (though few systems do truly incremental NLU). For simpler designs, it's fine to wait until end-of-speech, then send one HTTP request: audio -> text. The DM can be request-response as well (input text -> output text). The TTS might be request-response (text -> audio). The client app likely maintains a WebSocket to the server to send audio and receive audio (for low latency, as opening new HTTP connections for each audio stream could add overhead). Some open solutions like **DeepSpeech** have a streaming API, and projects like **Kaldi** have online decoding with websockets through variants like kaldinet. There are also specialized message brokers (MQTT, RabbitMQ, etc.) that can be used in IoT scenarios to send audio data and receive responses asynchronously, which is useful for edge deployments.

Finally, **error handling and fallback** need consideration in the architecture. If ASR fails to return anything (maybe due to noise or an unrecognized utterance), the system should handle that (e.g., ask the user to repeat). If NLU yields low confidence for an intent, the DM might trigger a clarification. These decisions can be part of the DM logic (for instance, Rasa has a fallback policy for low NLU confidence). The pipeline should propagate metadata too – e.g., the ASR could output a confidence score or alternative hypotheses; an advanced DM could utilize those (for example, "Did you say play **The Beatles**?" if it's unsure between Beatles and something else). Designing the interfaces with such information in mind can improve the robustness of the voice bot.

In summary, the pipeline architecture binds together ASR, NLU, DM, NLG, TTS into a cohesive system. The goal is to make this integration seamless so the user just experiences a fluid conversation. By using open-source components, you have the freedom to customize each link in the chain and optimize how they communicate. Whether it's a simple single-machine setup or a distributed cloud deployment, the principles remain: capture voice, transcribe to text, understand intent, decide response, generate text, synthesize voice – all as quickly and accurately as possible.

7. Performance and Optimization

Building a SOTA voice bot is not just about getting the components to work – they must work efficiently. **Latency** and **throughput** are critical. A voice assistant should ideally respond in under a second for a snappy experience; any latency above ~200–300 milliseconds in each stage can become noticeable to users (Source: developer.nvidia.com). There are several strategies to optimize performance across the pipeline:

- **Model Size vs. Speed Trade-offs:** Often there is a trade-off between accuracy and speed. For ASR, Whisper large might give the best accuracy but has a slower inference time than Whisper small or medium. You might choose a smaller model if it meets accuracy requirements to gain speed. Similarly, for NLU if you're using a Transformer for intent, a DistilBERT might be faster (with slightly lower accuracy) than a full BERT. Evaluate what model size truly suffices for your domain – sometimes a tiny model fine-tuned on in-domain data can outperform a huge generic model on that specific task.

- **Quantization and Model Optimization:** Converting neural network weights from float32 to int8 or int16 (quantization) can dramatically speed up inference and reduce memory, often with minimal accuracy loss. For example, quantizing an ASR acoustic model or a TTS model can improve CPU inference times, which is valuable if you deploy on commodity hardware (Source: play.ht). Tools like ONNX Runtime, TensorRT (NVIDIA), or open-source quantization libraries can help. Additionally, techniques like **pruning** (removing redundant neurons) or **knowledge distillation** (training a smaller model to mimic a larger one) are viable open-source strategies to shrink model sizes.

- **Streaming and Parallelism:** Utilize streaming where possible. A streaming ASR that processes audio on the fly can output partial results and effectively pipeline the processing (overlapping with the user's speech). This can cut perceived latency because by the time the user finishes talking, a transcript is nearly ready. On the TTS side, one could start playing the beginning of the audio while the rest is still being generated (some TTS architectures allow chunked

synthesis). Also consider parallelism: if your pipeline is microservice-based, the ASR, DM, and TTS could potentially run in parallel for different users. Within a single request, one can't parallelize sequential dependencies, but you can ensure, say, the TTS for user1 is synthesizing while user2's ASR is decoding – meaning having enough instances/threads to handle multiple interactions concurrently.

- **Hardware Acceleration:** GPUs or specialized accelerators (like NVIDIA TensorRT cores, Google TPUs for some models, or even smaller AI chips for edge) can drastically improve performance. For instance, running ASR and TTS on GPUs was shown to meet real-time requirements easily (Source: developer.nvidia.com). If you expect a high volume of traffic or need minimal latency, invest in hardware for the heavy parts (ASR and TTS typically). Also, if using CPU, make use of vectorized operations (like Intel MKL or ARM NEON optimizations). Open-source libraries often can detect and use these – e.g., PyTorch or TensorFlow will utilize AVX2/FMA on modern CPUs.

- **Batching and Caching:** In server scenarios, you might batch multiple requests to better utilize hardware, though this is more applicable to throughput than single-interaction latency. For example, if using a GPU ASR service, it could transcribe N audio streams in parallel as a batch for efficiency, but this introduces a slight delay to collect the batch. It's a trade-off: high throughput vs. low latency. For a voice assistant where latency is key, you usually process each request ASAP (no batching), except maybe in TTS if you want to generate multiple sentences together for efficiency. Caching can help if the system has repetitive queries – e.g., if many users ask the same question, caching the ASR result might not apply (audio varies), but caching an NLU or database query result or even a synthesized TTS audio for a common phrase ("I'm sorry, I didn't catch that") could avoid recomputation. Some voice bots cache the audio of frequent prompts to play them instantly.

- **Optimizing Dialogue Flow:** Performance isn't just raw compute – the design of conversations matters. If the bot's logic causes unnecessary turns, it feels sluggish to the user. For example, asking too many clarification questions will extend the overall interaction time. So, optimizing the **dialogue policy** (combining steps when possible, using confirmations smartly) improves the perceived performance of the system. This is more on the design side, but it's worth mentioning that a crisp dialogue that does in 2 turns what another bot does in 4 turns will feel faster and more "intelligent."

- **Monitoring and Benchmarking:** It's crucial to monitor latency of each component in real deployments. Tools can log how long ASR took for each utterance, how long NLU/DM took (which might be tiny, say 10ms, or more if using heavy NLU models), and TTS time. This helps

identify bottlenecks. Perhaps you find that 95% of the time is spent in ASR for long utterances – then focusing on speeding up ASR (or limiting input length) might be key. If TTS is the bottleneck, maybe use a faster vocoder or a lower sampling rate output.

- **Accuracy vs. Latency trade-offs:** There's also a balance between making the system *fast* and making it *correct*. Sometimes you might favor a slightly slower but more accurate ASR for complex language input to avoid misrecognitions that lead to error handling (which ultimately slows the conversation more). Or you might implement a two-pass approach: a quick first-pass ASR to get something out, and then a second-pass refine if needed (some systems do a quick greedy decode then a slower re-score for final output – though this is more complexity). The key is to ensure the user doesn't feel a lag. If an operation is slow but unavoidable (like a long external API call to fetch info), the DM can be optimized to handle that by playing a "typing indicator" or a brief "Let me check that for you..." prompt, which at least engages the user during the wait.

In conclusion, performance tuning is an ongoing effort. By using open-source tools, you have the advantage of transparency (you can dig into code to remove inefficiencies or adjust parameters). Measure the WER of your ASR and the latency; measure the accuracy of NLU on real queries; measure end-to-end conversation success rates. Then optimize bottlenecks: maybe it's as simple as using a GPU, or as involved as retraining a smaller model. The end goal is a voice bot that feels responsive and accurate, achieving an optimal blend of speed and intelligence. As one guide noted, the best systems must *"respond with the accurate answer in almost real time"*, as even a few hundred milliseconds added can hamper user experience (Source: developer.nvidia.com). Keep that ethos in your optimization efforts.

8. Evaluation and Benchmarking

To ensure your voice bot is truly state-of-the-art, rigorous evaluation of each component is essential. We need to measure how well the ASR is transcribing, how correctly the NLU is understanding, how effectively the DM is driving conversations, and the quality of TTS outputs. Here are the key metrics and methods for evaluating each part:

- **ASR Evaluation:** The primary metric for speech recognition is **Word Error Rate (WER)** (Source: jsaer.com). WER is the percentage of words that were incorrectly recognized, computed by comparing the ASR transcript to a human reference transcript (using Levenshtein distance to count substitutions, deletions, insertions). A lower WER means better accuracy. For example, a WER of 5% is excellent (95% words correct) on clean read speech, whereas casual

conversational speech might have higher WER. Other related metrics include **Character Error Rate (CER)** (useful for languages without clear word boundaries or as a proxy for phoneme errors) and **Sentence Error Rate** (percentage of whole sentences with any error). In research, you'll also see **PER (Phoneme Error Rate)** (Source: jsaer.com) for phonetic-level evaluation. To benchmark ASR, one should test on standard datasets: e.g., Librispeech test-clean/test-other for English, Common Voice for various languages, or domain-specific sets (like an internal dataset of actual user queries). Tools like NIST SCTK or asr-evaluation Python packages can calculate WER given reference and hypothesis texts. When evaluating, also consider conditions: how does WER change with background noise, different accents, spontaneous speech vs scripted? A comprehensive evaluation covers these scenarios. Achieving state-of-the-art results might mean comparing your ASR choice against known WER from literature (e.g., Whisper large has about 2.7% WER on Librispeech clean [1†], which is near human level).

- **NLU Evaluation:** NLU has two aspects – intent classification and entity extraction (also called slot filling). For intent classification, standard metrics are **accuracy** (percent of utterances where the top predicted intent matches the true intent) and sometimes **precision/recall/F1** if you consider multi-label or want to see per-intent performance. If an utterance can have multiple intents, then precision/recall become relevant (multi-intent is a bit advanced; many systems assume one intent per utterance). For entity extraction, you treat it like a named entity recognition task: use **Precision, Recall, and F1-score** for extracted entities, often computed for each entity type as well as overall (Source: topbots.com). For example, if the user said "Book flight from NYC to London" and the system extracted origin=NYC, destination=London, you check if those exactly match the ground truth entities. An entity is counted correct if both the type and the value span are correct. You might also compute **Slot Error Rate (SER)**, which is like WER for slots – the fraction of slots missed or wrong (Source: topbots.com). Another metric used in dialog literature is **Sentence-level Semantic Accuracy (SLSA)** – basically whether the intent and all required slots are correctly understood for the utterance (Source: topbots.com). Rasa provides evaluation utilities that output these metrics, and Snips NLU had a benchmarking library as well (Source: github.com). Benchmark datasets for NLU include ATIS (airline travel requests), SNIPS, MultiWOZ (which has multi-turn dialogs with intents and slots per turn), and CLINC150 (150 intent dataset). If you incorporate an open QA component (like Haystack), evaluating that part means measuring answer accuracy (e.g., exact match or F1 against known answers, similar to SQuAD evaluation).

- **Dialogue Management Evaluation:** Evaluating a dialogue manager is more complex because it's about the whole conversation. One common metric for task-oriented dialogues is **Task Success Rate** (Source: topbots.com) – did the dialogue achieve the user's goal? This often requires defining success criteria (e.g., user wanted to book a ticket, was a ticket booked by the

end?). It can be measured by having a set of dialogues either with a simulator or real users and marking if all information was obtained and correct outcome reached. Another is **Dialog Efficiency** – how many turns did it take, were there unnecessary exchanges? Typically measured in number of turns to completion (Source: topbots.com)(Source: topbots.com). A dialog that succeeds in 6 turns is more efficient than one that took 10 turns for the same task, for instance. There are also qualitative metrics: e.g., **appropriateness of responses**, or **user satisfaction** gathered via surveys. In research, automated metrics like **average reward** (if using reinforcement learning, where each dialog gets a reward signal) or the **Kappa coefficient on a confusion matrix** for task success have been used (Source: topbots.com). For open-domain chatbots, automatic metrics (BLEU, etc.) don't correlate well with human judgments (Source: topbots.com), so human evaluation remains key – e.g., having users rate dialogues on criteria like coherence, usefulness, etc. If focusing on task-oriented bots, you can construct a set of test conversations (even using a user simulator to run through various scenarios with the bot) to systematically measure success rate and any errors. The DSTC (Dialog System Technology Challenge) competitions provide some methodologies for evaluation, such as comparing the dialog state tracking accuracy (if your DM explicitly tracks state, you can measure how often it correctly tracks the user's intentions across turns) (Source: topbots.com).

- **NLG Evaluation:** For generated responses, if using templates, there's not much to evaluate aside from proofreading them and perhaps AB testing variations. But if using a generative NLG model, evaluation can be tricky. Automatic metrics from machine translation and summary tasks are sometimes employed: **BLEU, ROUGE, METEOR** compare the generated response to a reference response (Source: topbots.com), but in open conversation there can be many valid responses, so these scores often don't reflect true quality. More meaningful is **human evaluation** – have people rate the bot's responses for correctness, naturalness, and appropriateness. One specific metric for NLG naturalness is to do a MOS-like test (mean opinion score on a scale for how human-like the phrasing is). Another aspect is **factual correctness** (especially if your NLG or DM may hallucinate; ensure the content in the response is factually accurate w.r.t. your knowledge base). In task-oriented dialogue, you can measure NLG quality indirectly by task success (if the response was unclear, users won't be able to follow up correctly, etc.). If we refer to the survey by Deriu et al., they mention using **F1 score to measure the correctness of the NLG content** (i.e., did it include the necessary info) and BLEU/ROUGE for surface realization quality, plus human judgments for fluency (Source: topbots.com).

- **TTS Evaluation:** Text-to-Speech quality is often judged by **Mean Opinion Score (MOS)** tests (Source: zilliz.com), where human listeners rate audio samples (typically on a 1–5 scale). MOS is resource-intensive (needs a panel of listeners in quiet conditions), but it is the gold standard for

naturalness and overall quality. If you don't have the means for a MOS test, you can subjectively listen and compare to references. There are some objective metrics: **Mel-Cepstral Distortion (MCD)** measures the difference between generated audio and a reference recording of the same sentence (Source: zilliz.com) – lower MCD (in dB) is better and correlates somewhat with quality. **PER (Phone Error Rate)** or using an ASR to transcribe the TTS output and computing WER against the input text (sometimes called intelligibility score) is also done – if a TTS is good, an ASR should be able to transcribe it perfectly. There's also **PESQ (Perceptual Evaluation of Speech Quality)** and **STOI (Short-Term Objective Intelligibility)** which are metrics from telecommunications, used to evaluate voice quality objectively (Source: zilliz.com). These can be used as rough proxies; for example, PESQ predicts a MOS score by algorithm. In practice, for a voice bot, ensure the TTS is intelligible (no words are mispronounced) and pleasant. You might benchmark different voices or engines by synthesizing a fixed set of test sentences (some short, some long, various phonetic content) and either doing a listening test or using the above metrics. Additionally, test the TTS in real conversation context – sometimes a voice might be clear in isolated sentences but sound monotonic or awkward across an interactive dialogue. Fine-tuning the prosody (via SSML tags for pauses, etc.) might improve the perceived quality.

- **End-to-End Evaluation:** Finally, consider end-to-end performance of the entire voice bot. This can be done with **scenario testing** – define sample user tasks and have either human testers or a user simulator go through them with the bot. Measure overall success, user satisfaction, and identify failure points. For a comprehensive benchmark, you could measure: first-turn success rate (does the bot understand user on first try), average number of reprompts needed, and user ratings. In academia, end-to-end dialog is sometimes evaluated by having humans chat with the bot and then rate the interaction on scales like 1-5 for various attributes (as seen in the Alexa Prize challenge for open-domain chatbots).

By continuously evaluating with these metrics, you can track improvements and regressions. For example, if you update the ASR model, see if WER went down. If you tweak NLU, check if intent accuracy improved. Use confusion matrices to see which intents are getting confused and perhaps add training data for them. Evaluation tools exist in open-source form: Rasa has `rasa test` for NLU and stories, which produces reports. There are also suites like **Dialeval** for dialogues. It's also good to benchmark against other systems if possible – e.g., how does your open-source voice bot stack up against a service like Google Dialogflow or Alexa on similar tasks? This can highlight areas to focus on.

In essence, a state-of-the-art bot not only uses strong components but also quantifies their performance rigorously. Metrics like WER, F1, success rate, and MOS give you evidence of quality (Source: jsaer.com)(Source: topbots.com). Backed with this data, you can iterate on the system to

close any gaps and ensure that each part of the pipeline meets the desired standards for accuracy and user experience.

9. Deployment Options (On-Premises, Cloud, and Edge)

Deploying a voice bot in a real-world environment requires planning how the system will be hosted and scaled. Open-source solutions offer flexibility in deployment: you can run everything fully on-premises (offline, on your own hardware), in the cloud (using virtual machines or containers in a cloud provider), or on edge devices (embedded hardware or user devices), or a hybrid of these. We will discuss each approach and the relevant open-source deployment technologies like Docker and Kubernetes for containerization and orchestration.

On-Premises Deployment: Deploying on-prem means all components of the voice bot run in your organization's local servers or data centers (or even a single machine in an offline environment). This is often chosen for privacy, security, or latency reasons – for example, a bank may want a voice assistant that **runs locally to ensure data privacy** (Source: rasa.com), so no audio or text leaves the premises. With open-source tools, on-prem is straightforward since you're not tied to a vendor's cloud. You would set up servers with the necessary environment (e.g., Linux machines with GPU cards for ASR/TTS if needed). Each component (ASR, NLU, etc.) can run as a service or process on those machines. Using Docker containers is highly beneficial here: you can containerize each component to isolate dependencies and simplify installation. For instance, you might use the official Rasa Docker image for the NLU/DM, a DeepSpeech or Vosk Docker image for ASR, etc. On-prem deployments give you full control – you can even run without internet access (useful for secure facilities or IoT scenarios). Tools like **Docker Compose** can help define a multi-container setup on a single host (for development or small-scale on-prem deploy). For larger on-prem deployments (multiple servers), Kubernetes can be installed on-prem (either vanilla Kubernetes or a distribution like Red Hat OpenShift, or even lightweight K3s for small clusters) to coordinate containers across servers. Running on-prem also allows integration with internal systems directly – for example, connecting the bot to internal databases or APIs without exposing them externally.

Cloud-Native Deployment: Here, you deploy the voice bot on cloud infrastructure (AWS, Azure, GCP, or private cloud). You still use open-source components, but you leverage the cloud provider's hardware and services. A common approach is containerization + orchestration: *"Docker and Kubernetes"* are practically the standards for cloud-native deployment of complex apps. You would create Docker images for each service if not already available. Many open-source projects already provide images – e.g., *"Rasa hosts many pre-built Docker containers on Docker Hub"* for different

versions (Source: learning.rasa.com), and projects like Vosk have Docker images per model for easy deployment in Kubernetes (Source: jmrobles.medium.com). After containerizing, you define how they communicate (e.g., maybe a Kubernetes Service for ASR, one for NLU/DM, one for TTS). You might use a **Helm chart** if available (Rasa has official helm charts for deploying on K8s). In the cloud, you can also take advantage of auto-scaling: for instance, if load increases (say many concurrent users), Kubernetes can spin up more pods of the ASR service to handle it. Cloud deployment also makes it easier to distribute globally – you could run instances in multiple regions to serve users with low latency. When deploying in the cloud, ensure you secure everything (more in Security section) since your endpoints could be accessible publicly. Use cloud features like virtual networks so that internal communication between services isn't exposed. A typical cloud setup: use a Kubernetes cluster (managed k8s like GKE/EKS or self-managed) – deploy ASR, NLU/DM, TTS as microservices. You might also deploy a gateway service that the client apps talk to, which then fans out to these internal services. For stateful components like if Rasa needs a tracker store or logs, you might use cloud storage or databases (PostgreSQL for Rasa tracker for example) – those too can be open-source and self-hosted or use the cloud provider's managed DB for convenience. Cloud deployment, while not *intrinsically* required by an open-source bot, is often chosen for ease of scaling and maintenance. You can also use container orchestration to roll out updates with zero downtime, etc.

Edge and Embedded Deployment: In some cases, the voice bot (or parts of it) runs on edge devices – meaning on the device that is physically near the user or part of a product. This could be a smart speaker, a smartphone, a car's onboard computer, or an IoT device like a Raspberry Pi. Open-source voice assistants like **Mycroft AI** and **Rhasspy** are designed for edge use, combining components that can run on a single board computer. For edge deployment, typically the ASR and TTS models need to be lightweight (since edge devices have limited CPU, maybe no GPU). Tools like Vosk are specifically cited to work "on edge devices… with a small model size fit for mobile phones or IoT" (Source: medium.com). For example, you can run a 50 MB Vosk model on an Android device to do offline recognition. There's also **Piper TTS** which is optimized for Raspberry Pi, allowing offline TTS with good quality. Edge deployment can be fully offline (no network needed) – great for privacy and working in environments without internet. However, edge devices might not handle heavy ML for large models, so often a compromise is made: either use smaller models (with possibly slightly lower accuracy) or use a hybrid approach (do wake word detection and maybe some basic commands on-device, but offload complex requests to a server). With open source, hybrid schemes are possible: e.g., a device could have a hotword detector (Porcupine or Precise), use a local small ASR for known commands, and if it doesn't understand, send the audio to a more powerful server ASR for processing – all of which you can set up without proprietary services.

Containerization & Orchestration Details: Docker is practically a must for ease of deployment. You'd create a Dockerfile for any custom component (say you wrote a custom DM server), and otherwise use existing images for standard components (like PostgreSQL, RabbitMQ if you use one, Rasa's image, etc.). Inside Docker, you can pack all necessary models (though be mindful of image size). With Kubernetes, you define deployments for each microservice and perhaps use a message broker or HTTP calls between them. For example, one Medium post shows how to *"put Vosk STT service in Kubernetes"*, noting that *"the project already has Docker images for each language"* and demonstrating a YAML config (Source: jmrobles.medium.com). Kubernetes brings benefits like self-healing (if a container crashes, it restarts), scaling, and easy routing. If using Kubernetes for a voice bot, one challenge is real-time audio streaming – ensure low network latency and maybe use protocols like gRPC which work nicely in k8s. You might also consider using a **service mesh** if the architecture grows complex, but that might be overkill initially.

Deployment Strategy Examples: Suppose we have a voice bot for customer support. An on-prem deployment might involve a server at the call center running the voice bot; calls are routed via SIP to an ASR service on that server which transcribes and feeds to the bot, and the bot's answer is TTSed back to the phone – completely internal, ensuring no call leaves the company network. A cloud deployment of the same might instead use a cloud Speech-to-text instance (or an open source ASR on a cloud VM) scaling to many lines, and the company doesn't maintain hardware. An edge example: a voice-enabled appliance (like a smart fridge) running a small-footprint voice assistant inside it so that it works even offline.

Finally, consider **DevOps practices**: use CI/CD to build your Docker images whenever you update code, use infrastructure-as-code (like Terraform or k8s YAML) to manage deployment configurations. Open-source orchestration means you are not locked in – you could move from one cloud to another or to on-prem since everything is in portable containers.

In summary, open-source voice bots are extremely flexible in deployment. Whether you need the privacy of fully on-prem, the scalability of cloud, or the low-latency offline capabilities of edge devices, you can do it. Utilizing Docker and Kubernetes, you ensure your solution is portable and scalable: *containerization* encapsulates each service with its dependencies, and *orchestration* handles running those containers across the infrastructure. With options like these, one can deliver a voice assistant that runs wherever it's needed – from data centers to living rooms – using purely open technology.

10. Security and Privacy

Security and privacy are paramount in voice bot applications, especially since they involve processing potentially sensitive user data (people might speak personal information, account details, health queries, etc.). Using open-source components gives you full control over data handling, which can be a big advantage for privacy – but it also puts the responsibility on you to implement best practices. Let's break down key considerations and how to address them with open-source best practices:

Data Handling and Privacy Compliance: Voice data (both audio and transcriptions) should be treated as sensitive personal data. If deploying on-prem or on-device, you keep data locally, greatly reducing exposure. If any data is sent over networks (e.g., a mobile app sending audio to a server), ensure it's encrypted in transit (TLS/HTTPS or WSS for WebSocket). Also encrypt sensitive data at rest – if you store conversation logs or audio recordings, use disk encryption or at least restrict access. An open-source voice bot can be designed to avoid storing data unnecessarily: for example, you might choose not to log raw audio at all, or to anonymize transcripts (removing phone numbers, names) if kept for analytics. **Privacy regulations** like GDPR require giving users some control – e.g., the ability to delete their data. So, have a mechanism to delete conversation records associated with a user upon request if you store them. Since you're using open-source, you won't be sending data to third-party AI services, which helps compliance (no surprise data sharing). In contexts like healthcare or finance, ensure you follow standards (HIPAA in US healthcare, for instance) – this may involve running everything locally (no cloud) and strong access controls.

User Consent and Transparency: Often overlooked, but let users know that they are being recorded or their voice is processed. If it's an app, have a clear privacy policy. If it's a physical device, an audible tone or indicator when listening (like Alexa's blue ring) is a good practice so users know when the mic is active. For wake-word based devices, ensure the wake word detection (which can be open-source like Mycroft Precise or Porcupine) is reliable to avoid inadvertently recording when not intended, as that becomes a privacy issue.

Secure Communication: Use secure channels for all inter-component communication, especially if distributed. For example, if your ASR service and DM service talk over HTTP, make it HTTPS (even inside a cloud VPC, it's wise to encrypt to defend against any internal threats). Use authentication between services – e.g., an API token or key for the client app to call your bot's API, so that only authorized sources can send audio and receive data. In a Kubernetes deployment, one might use

mutual TLS between microservices (service mesh can help, or simpler, run all in a private network and ensure no external access). If a public endpoint is needed (say a WebSocket for a web client to stream audio), protect it with authentication (JWT, API keys) to prevent misuse or eavesdropping.

Access Control and Logging: Within your system, not everyone should have access to everything. For example, if you have a database of transcripts, restrict who (which microservices or which administrators) can read it. Use roles – an admin might see text transcripts for improving the model, but maybe they shouldn't see raw audio unless needed. If using a web management interface (like Rasa X or Botpress's admin UI), secure it with strong passwords and ideally behind a VPN if possible. Botpress notes that it offers *"built-in security features such as access control and encryption"*, whereas with Rasa *"you manage your own encryption, data storage, and privacy compliance"* when self-hosting (Source: keencomputer.com). In other words, open-source gives flexibility but you must implement controls: e.g., you might add basic auth to your REST endpoints, or use an API gateway like Kong or Traefik with access control to front your bot services.

Vulnerabilities and Updates: Keep your open-source components up to date with security patches. Just as any software, an open-source library could have a vulnerability. For instance, an outdated web server in your bot's stack might be exploitable. Use dependency scanning tools (OWASP dependency-check, etc.) to spot known CVEs in the libraries you use. Regularly update Docker base images to include security patches. For any custom code, follow secure coding practices since a voice bot could be a target (imagine someone tries to send a malicious payload in an utterance to exploit the system – e.g., if you improperly handle certain inputs, it might cause crashes or worse).

Secure Model and Data Storage: The ML models themselves (the files for ASR, NLU, etc.) should be stored securely to prevent tampering. An attacker altering a model could, in theory, change behavior (like a poisoned model). Store model files with correct permissions (non-world-readable on Linux, etc.). If you distribute the bot (say an edge device in users' homes), consider the risk of someone extracting or manipulating the model (this is more of an anti-tamper concern; maybe not critical unless the model itself encodes sensitive info which typically it doesn't in open usage).

Preventing Data Leakage: Because you control the stack, ensure that none of the open-source components are inadvertently sending data externally. Some cloud-based SDKs do that, but pure open-source ones like Kaldi, Rasa, etc., generally don't phone home. Double-check configurations – e.g., disable any analytics in these tools if present. Also, if your system uses any third-party services (like for SMS or something), consider what data is sent and ensure it's minimized and secured.

Privacy-Preserving Machine Learning: This is an advanced topic, but worth a mention. Techniques like **differential privacy** and **federated learning** can be applied if you plan to improve models using user data. For instance, if you gather conversation logs to improve the NLU, you could anonymize them and strip personal identifiers. Differential privacy (adding noise to ensure individuals can't be re-identified from aggregated data) is something to consider if doing large-scale analytics on user queries.

Ethical and Security Concerns: Voice bots can be targets of misuse – e.g., an attacker might try to trigger certain actions by imitating the user's voice or by finding phrases that fool the NLU. Implement authentication for actions that need it (if the bot is doing something sensitive like transferring money, don't rely solely on voice; consider a secondary auth factor or at least a confirmation). Also, include throttling/rate limiting – to prevent DDoS or abuse, limit how many requests a single IP or user can make in a short time, etc., which can be done at the web server or gateway level.

Audit and Monitoring: Maintain logs of access and actions (with caution to not log sensitive info in plaintext). Monitor these for unusual activity. For example, if someone is repeatedly trying to invoke an admin-only intent via voice, that might be an attempted exploit. With open-source, you can integrate with SIEM tools easily by outputting standard logs. If an incident happens, you have all the pieces (since you host them) to investigate; whereas with a closed cloud service you might not get that detail.

To sum up, leveraging open-source means you *own the responsibility* for security and privacy, but you also *own the control*. Best practices include encrypting data in transit and at rest, limiting access (both network access and user access) to the system, and handling user data in compliance with regulations. As the JSAER research paper highlighted, *"Protecting user privacy and preventing unauthorized access requires secure data handling practices."* (Source: jsaer.com) This includes everything from secure coding to infrastructure security. Additionally, by self-hosting, *"organizations can manage their own encryption, data storage, and privacy compliance"*, avoiding the risks of third-party cloud exposure (Source: keencomputer.com). Implementing these measures ensures that your state-of-the-art voice bot not only performs well, but also maintains the trust and safety that users and regulators demand.

Sources: The information in this report has been gathered from a range of up-to-date resources, including technical blogs, research papers, and documentation of the mentioned open-source projects. Key sources include a Deepgram study on open ASR models (Source: deepgram.com) (Source: deepgram.com), a Medium comparison of ASR engines (Source: medium.com), Rasa's official blog and documentation for NLU/DM insights (Source: botfriends.de)(Source:

keencomputer.com), open-source TTS benchmarks (Source: tderflinger.com), as well as NVIDIA's guide on deploying Rasa with Riva for architecture considerations (Source: developer.nvidia.com), among others. All citations are provided in-text in the format source†lines for reference. Each component and recommendation described is grounded in these sources, ensuring that this guide reflects state-of-the-art practices as of 2025.

Tags: voice bot, open source, asr, speech recognition, openai whisper, kaldi, vosk, speech to text

---

About ClearlyIP

ClearlyIP Inc. — Company Profile (June 2025)

---

1. Who they are

ClearlyIP is a privately-held unified-communications (UC) vendor headquartered in Appleton, Wisconsin, with additional offices in Canada and a globally distributed workforce. Founded in 2019 by veteran FreePBX/Asterisk contributors, the firm follows a "build-and-buy" growth strategy, combining in-house R&D with targeted acquisitions (e.g., the 2023 purchase of Voneto's EPlatform UCaaS). Its mission is to "design and develop the world's most respected VoIP brand" by delivering secure, modern, cloud-first communications that reduce cost and boost collaboration, while its vision focuses on unlocking the full potential of open-source VoIP for organisations of every size. The leadership team collectively brings more than 300 years of telecom experience.

---

2. Product portfolio

- **Cloud Solutions** – Including *Clearly Cloud* (flagship UCaaS), **SIP Trunking**, **SendFax.to** cloud fax, **ClusterPBX OEM**, **Business Connect** managed cloud PBX, and **EPlatform** multitenant UCaaS. These provide fully hosted voice, video, chat and collaboration with 100+ features, per-seat licensing, geo-redundant PoPs, built-in call-recording and mobile/desktop apps.
- **On-Site Phone Systems** – Including CIP PBX appliances (FreePBX pre-installed), ClusterPBX Enterprise, and Business Connect (on-prem variant). These offer local survivability for compliance-sensitive sites; appliances start at 25 extensions and scale into HA clusters.
- **IP Phones & Softphones** – Including CIP SIP Desk-phone Series (CIP-25x/27x/28x), fully white-label branding kit, and *Clearly Anywhere* softphone (iOS, Android, desktop). Features zero-touch provisioning via Cloud Device Manager or FreePBX "Clearly Devices" module; Opus, HD-voice, BLF-rich colour LCDs.

- **VoIP Gateways** – Including Analog FXS/FXO models, VoIP Fail-Over Gateway, POTS Replacement (for copper sun-set), and 2-port T1/E1 digital gateway. These bridge legacy endpoints or PSTN circuits to SIP; fail-over models keep 911 active during WAN outages.
- **Emergency Alert Systems** – Including **CodeX** room-status dashboard, **Panic Button**, and **Silent Intercom**. This K-12-focused mass-notification suite integrates with CIP PBX or third-party FreePBX for Alyssa's-Law compliance.
- **Hospitality** – Including **ComXchange** PBX plus PMS integrations, hardware & software assurance plans. Replaces aging Mitel/NEC hotel PBXs; supports guest-room phones, 911 localisation, check-in/out APIs.
- **Device & System Management** – Including **Cloud Device Manager** and **Update Control (Mirror)**. Provides multi-vendor auto-provisioning, firmware management, and secure FreePBX mirror updates.
- **XCast Suite** – Including Hosted PBX, SIP trunking, carrier/call-centre solutions, SOHO plans, and XCL mobile app. Delivers value-oriented, high-volume VoIP from ClearlyIP's carrier network.

---

3. Services

- **Telecom Consulting & Custom Development** – FreePBX/Asterisk architecture reviews, mergers & acquisitions diligence, bespoke application builds and Tier-3 support.
- **Regulatory Compliance** – E911 planning plus **Kari's Law**, **Ray Baum's Act** and **Alyssa's Law** solutions; automated dispatchable location tagging.
- **STIR/SHAKEN Certificate Management** – Signing services for Originating Service Providers, helping customers combat robocalling and maintain full attestation.
- **Attestation Lookup Tool** – Free web utility to identify a telephone number's service-provider code and SHAKEN attestation rating.
- **FreePBX® Training** – Three-day administrator boot camps (remote or on-site) covering installation, security hardening and troubleshooting.
- **Partner & OEM Programs** – Wholesale SIP trunk bundles, white-label device programs, and ClusterPBX OEM licensing.

---

4. Executive management (June 2025)

- **CEO & Co-Founder: Tony Lewis** – Former CEO of Schmooze Com (FreePBX sponsor); drives vision, acquisitions and channel network.
- **CFO & Co-Founder: Luke Duquaine** – Ex-Sangoma software engineer; oversees finance, international operations and supply-chain.
- **CTO & Co-Founder: Bryan Walters** – Long-time Asterisk contributor; leads product security and cloud architecture.

- **Chief Revenue Officer: Preston McNair** – 25+ years in channel development at Sangoma & Hargray; owns sales, marketing and partner success.
- **Chief Hospitality Strategist: Doug Schwartz** – Former 360 Networks CEO; guides hotel vertical strategy and PMS integrations.
- **Chief Business Development Officer: Bob Webb** – 30+ years telco experience (Nsight/Cellcom); cultivates ILEC/CLEC alliances for Clearly Cloud.
- **Chief Product Officer: Corey McFadden** – Founder of Voneto; architect of EPlatform UCaaS, now shapes ClearlyIP product roadmap.
- **VP Support Services: Lorne Gaetz** (appointed Jul 2024) – Former Sangoma FreePBX lead; builds 24×7 global support organisation.
- **VP Channel Sales: Tracy Liu** (appointed Jun 2024) – Channel-program veteran; expands MSP/VAR ecosystem worldwide.

---

5. Differentiators

- **Open-Source DNA:** Deep roots in the FreePBX/Asterisk community allow rapid feature releases and robust interoperability.
- **White-Label Flexibility:** Brandable phones and ClusterPBX OEM let carriers and MSPs present a fully bespoke UCaaS stack.
- **End-to-End Stack:** From hardware endpoints to cloud, gateways and compliance services, ClearlyIP owns every layer, simplifying procurement and support.
- **Education & Safety Focus:** Panic Button, CodeX and e911 tool-sets position the firm strongly in K-12 and public-sector markets.

In summary

ClearlyIP delivers a comprehensive, modular UC ecosystem—cloud, on-prem and hybrid—backed by a management team with decades of open-source telephony pedigree. Its blend of carrier-grade infrastructure, white-label flexibility and vertical-specific solutions (hospitality, education, emergency-compliance) makes it a compelling option for ITSPs, MSPs and multi-site enterprises seeking modern, secure and cost-effective communications.

---

information independently. All product names, trademarks, and registered trademarks mentioned are property of their respective owners and are used for identification purposes only. Use of these names does not imply endorsement. This document does not constitute professional or legal advice. For specific guidance related to your needs, please consult qualified professionals.