Offline Voice Assistant - Designing and Programming the Internet of Things

Andrea Leoni
University of Trento
Trento, Italy
andrea.leoni-1@studenti.unitn.it

***Abstract*—This work presents a fully offline voice assistant implemented as an IoT edge node with built-in observability. The system runs an end-to-end speech pipeline—speech-to-text with Whisper Tiny, on-device language understanding and generation with TinyLLaMA, and text-to-speech using Coqui TTS—entirely on local hardware. Audio is captured with silence detection and language-model outputs are streamed and synthesized in small chunks, enabling responsive, real-time interaction. At the IoT layer, components exchange events and telemetry over MQTT, are orchestrated with Node-RED, and persist data to SQLite (operational events) and InfluxDB (time-series metrics), which in turn drive dashboards and fault diagnosis. Measurements on commodity hardware show practical initialization and processing times, while telemetry summaries confirm stable throughput and graceful recovery under simulated broker/runtime faults. The results indicate that a low-latency conversational assistant can be treated as an instrumented IoT edge application rather than a black-box ML component.**

I. Introduction

Automated voice assistants are increasingly embedded in everyday devices, from smartphones to household appliances and embedded systems. Advances in hardware and software now allow sophisticated applications to run on small, energy-efficient platforms, offering hands-free interaction via natural speech. Public and private organizations also rely on conversational interfaces to support customer service, streamline workflows, and improve accessibility.

Mainstream systems such as Google Assistant, Amazon Alexa, and Apple Siri remain heavily dependent on cloud infrastructure. This dependence introduces three well-known challenges: (1) **privacy risks**, as user audio must be transmitted to external servers; (2) **latency**, due to network traversal and remote processing; and (3) **availability limitations**, since functionality degrades in low-connectivity or offline scenarios. These constraints motivate local, on-device assistants that preserve usability while operating entirely without Internet access.

This work targets the design and implementation of an *offline voice assistant* that minimizes hardware requirements while maintaining real-time, coherent, and interactive behavior. The assistant transcribes spoken prompts, processes them with a local language model, and synthesizes responses as natural-sounding speech, without reliance on external services. The key requirements guiding the design are:

- **Lightweight deployment:** Models and dependencies should fit within modest storage and memory budgets to enable installation on constrained devices (e.g., robots or single-board computers).
- **Connectivity-independent edge operation:** All speech processing and inference runs locally on the device, so the assistant keeps working even in the absence of internet connectivity; privacy is a natural side effect of this design.
- **Low latency:** End-to-end delay between user input and synthesized audio should support natural, interactive use.

From an Internet of Things perspective, the assistant is treated as an edge node with one sensor (microphone), one actuator (speaker) and a telemetry channel built on MQTT. Events and metrics are collected by Node-RED and stored in SQLite and InfluxDB, enabling dashboards and remote supervision typical of IoT deployments.

To satisfy these requirements, the system integrates SpeechRecognition for input capture with silence-based termination, Whisper Tiny for speech-to-text transcription, TinyLLaMA [1] for on-device conversational inference, and Coqui TTS (Tacotron2-DDC + HiFi-GAN v2) for text-to-speech synthesis. A streaming response pipeline converts partial language-model output into short audio fragments and plays them immediately, reducing perceived latency.

Beyond the speech pipeline, the assistant is instrumented as an *IoT edge node*: components publish events and metrics over a local MQTT broker; Node-RED orchestrates ingestion and fan-out; SQLite records immutable operational events; and InfluxDB collects time-series telemetry for monitoring. This architecture enables observability, diagnostics, and basic reliability features while maintaining offline design. A reproducible setup is provided via an automated installer and a dedicated Node-RED *Installer* tab for one-click environment preparation, verification, and execution.

In Section II, related work defining the state of the art is reviewed. Section III details the methodology and implementation, including the offline speech pipeline and the IoT observability layer. Section IV presents quantitative performance, telemetry-based metrics, and reliability observations. Section V concludes with implications and future directions.

II. Related Work

The development of offline voice assistants has gained momentum as privacy concerns, latency reduction, and connectivity constraints drive the need for on-device processing. Traditional commercial assistants, such as Google Assistant, Amazon Alexa, and Apple Siri, offer advanced speech recognition and natural language understanding (NLU) capabilities but rely heavily on cloud infrastructure, raising privacy issues and limiting usability in low-connectivity environments [2].

Gupta et al. [2] presented a customizable AI-powered voice assistant that integrates speech-to-text (STT), natural language processing, and text-to-speech (TTS) in a modular architecture. Their system employs Google Speech-to-Text and CMU Sphinx to enable both online and offline modes, enhancing robustness against ambient noise while allowing user-defined commands. The design emphasizes low latency, privacy, and adaptability, demonstrating that lightweight NLP models can offer competitive performance for essential tasks without cloud dependency.

Lazzaroni et al. [3] proposed an end-to-end embedded voice assistant targeting the Italian language, focusing on vehicular applications where low latency, privacy, and robustness are critical. The architecture integrates wake-word detection, automatic speech recognition (ASR), NLU, and TTS, optimized for deployment on NVIDIA Jetson AGX Xavier hardware. Using transfer learning from English ASR models (QuartzNet 15x5), the system achieves word error rates comparable to cloud solutions, while MelGAN and Tacotron2 ensure natural-sounding offline TTS. This work demonstrates that a fully offline stack can support multilingual and domain-specific use cases with minimal performance trade-offs.

Bermuth et al. [4] introduced *Jaco*, a privacy-focused offline assistant capable of running on low-resource devices such as the Raspberry Pi. The architecture of Jaco separates lightweight ”satellites” for audio capture and output from a central processing node, with all modules containerized for portability. The system supports skill-based extensibility, multilingual operation, and fine-grained permission control to protect user data. Benchmarking shows that Jaco outperforms many commercial and open-source solutions in specific domains, particularly under medium and low noise conditions.

A. Summary of Trends and Gaps

Across these works, several trends emerge:

1) **Shift toward fully offline processing** to address privacy and latency concerns;
2) **Modular architectures** that allow easy replacement of components (e.g. ASR, NLU, TTS) and adaptation to different domains;
3) **Skill-based extensibility** to enable user-defined commands and integration with external systems;
4) **Transfer learning for multilingual support**, allowing adaptation to languages with limited training data.

However, challenges remain in achieving cloud-level NLU accuracy in noisy environments, handling low-resource languages, and ensuring smooth integration across the speech pipeline on constrained hardware.

B. Positioning of the Present Work

The system implemented in this work—an **Offline Voice Assistant for Public Spaces**—builds directly on these research directions while introducing several practical engineering contributions. The architecture is entirely local, with no internet dependency during runtime, and integrates:

- **Wake-word detection** and STT using `pywhispercpp` with the Whisper Tiny model for lightweight and real-time transcription;
- **On-device LLM inference** using `llama-cpp-python` with TinyLLaMA-1.1B, optimized for fast and low-memory conversational response generation;
- **Local TTS synthesis** using Coqui TTS (Tacotron2-DDC + HiFi-GAN v2 vocoder), allowing natural speech output without cloud calls;
- **Streaming response pipeline** where the partial outputs of the LLM are converted to chunks of audio and queued for immediate playback, reducing perceived latency.

This design directly addresses the gaps identified in previous work by coupling a fully offline conversational pipeline with near-real-time responsiveness and low computational overhead, making it suitable for interactive installations in public environments where network connectivity, privacy, and user engagement are critical factors.

III. Methodology and Implementation

A. Environment Setup & Node-RED Installer Tab

A reproducible, offline-first setup is provided via a Python installer and a dedicated Node-RED *Installer* tab. The installer automates virtual environment creation, dependency installation, model downloads, and initialization of local storage artifacts (SQLite). The Node-RED tab exposes one-click actions to execute the installer and display progress logs.

- **Automated installer (Python)**. The script `install_assistant.py` creates a virtual environment, upgrades `pip`, installs a CPU-only PyTorch build, and the runtime dependencies. Then it downloads the Whisper Tiny model, a TinyLLaMA GGUF checkpoint, the Coqui TTS models, and initializes the SQLite database at `data/assistant.db`;
- **Invocation**. The installer can be launched from the command line or via Node-RED;
- **Startup**. After installation, the assistant entry point is `main.py` (callable from CLI or Node-RED Exec);
- **Node-RED ”Installer” tab**. A dedicated dashboard tab labeled *Installer* exposes a single-click installation path. It wires UI controls to an `exec` node configured to run a shell (`cmd.exe` on Windows) with the installer command; a text/log widget streams the subprocess output for traceability. It also provides a simple SQL script that creates the required tables for SQLite;
- **Artifacts and idempotency**. The installer is idempotent with respect to model files and the database: existing artifacts are reused. This behavior allows for repeated execution from the dashboard without side effects, which is convenient for demonstrations and for recovering partially completed setups.

B. Model Selection & On-Device Constraints

The assistant targets CPU-only edge hardware with strict limits on storage, memory, and latency. Model selection was therefore guided by the need to balance accuracy with a small footprint and low end-to-end delay in a fully offline stack.

1) Speech-to-Text (STT): Whisper Tiny is employed via `pywhispercpp` to transcribe recorded WAV prompts and to perform wake-word detection reusing the same ASR pass. This choice minimizes disk footprint and avoids a separate wake-word engine while preserving acceptable robustness in noisy environments. In the implementation, `transcribe()` aggregates model segments and publishes the transcript on MQTT for downstream components.

2) Language Model (LLM): A compact TinyLLaMA (1.1B, GGUF) model is used through `llama-cpp-python`. Responses are streamed token-by-token and buffered until punctuation to yield short phrases, reducing perceived latency and enabling overlapped TTS synthesis and playback.

3) Text-to-Speech (TTS): Coqui TTS with Tacotron2-DDC (spectrogram) and HiFi-GAN v2 (vocoder) synthesizes natural-sounding audio with low delay, while remaining compatible with CPU-only deployment. The synthesized chunks are queued for immediate playback to overlap compute with I/O.

4) Installer binding and artifact sizes: The installer provisions a CPU-only PyTorch wheel, the TinyLLaMA GGUF checkpoint, Whisper Tiny weights, and Coqui models, and initializes the SQLite store. This ensures reproducibility and a one-click setup from the Node-RED Installer tab.

5) Rationale summary: These selections jointly minimize memory and disk usage while enabling streaming interaction: the ASR handles both wake-word and prompt transcription; the LLM streams phrases as soon as punctuation is reached; and the TTS synthesizes short chunks for immediate playback. This configuration was validated in the evaluation chapter, where the STT and TTS stages dominate compute yet remain within real-time bounds on laptop-class CPUs; streaming mitigates perceived latency.

C. Pipeline Overview

The assistant is implemented as an on-device speech pipeline (microphone - STT - LLM - TTS) integrated with an IoT messaging and monitoring layer. The microphone listener captures audio until silence is detected and persists it in a temporary WAV file; resource usage is monitored via a decorator that logs CPU/RAM averages and peaks during each critical stage. Transcription produces a text prompt for the LLM, whose response is streamed in short phrases; each phrase is synthesized to speech and queued for playback. Throughout the flow, state and content events are published on MQTT topics (`assistant/state`, `assistant/stt`, `assistant/llm_chunk`) to support dashboards and storage.

D. Modules and Responsibilities

1) Microphone Listener: Audio is captured at 16kHz until silence is detected. The function `record_audio` writes a temporary WAV and reports timing and resource usage via the monitoring decorator (`monitor_resources`, label ”SR”).

2) Speech-to-Text (STT): Transcription aggregates model segments into a single string and publishes the transcript to `assistant/stt`. A simple wake-word detector operates on a low-cost transcription pass, checking for the token ”wake up”. STT execution is profiled via the monitoring decorator (label ”STT”).

3) LLM Engine: A compact local model is prompted with a fixed system message and the user transcript. Responses are streamed token-by-token and buffered until punctuation is observed and a minimum chunk length is reached; each completed chunk is emitted to `assistant/llm_chunk` and yielded to the TTS stage. The LLM function is profiled via the monitoring decorator (label ”LLM”). Constants `MAX_TOKENS` and `MIN_BUFFER_LENGTH` bound generation and chunk size respectively.

4) Text-to-Speech (TTS) and Audio Queue: Each phrase is synthesized to a WAV file and queued for playback. The background audio player publishes `assistant/state = speaking`, plays the audio, and removes the temporary file;

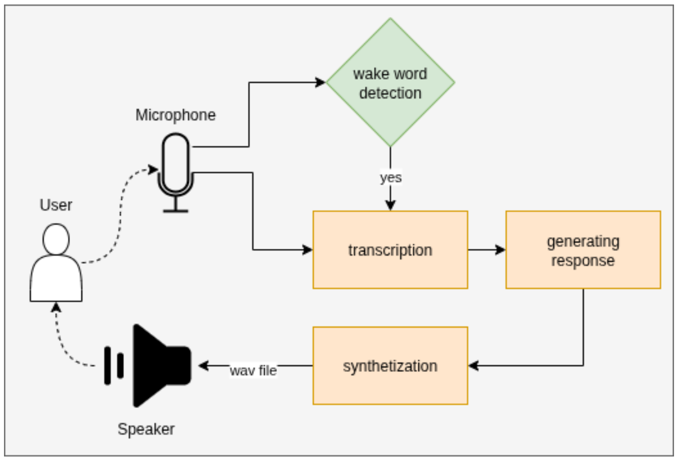

Fig. 1. End-to-end interaction between user and assistant: audio capture, transcription, language modeling, synthesis, and feedback.

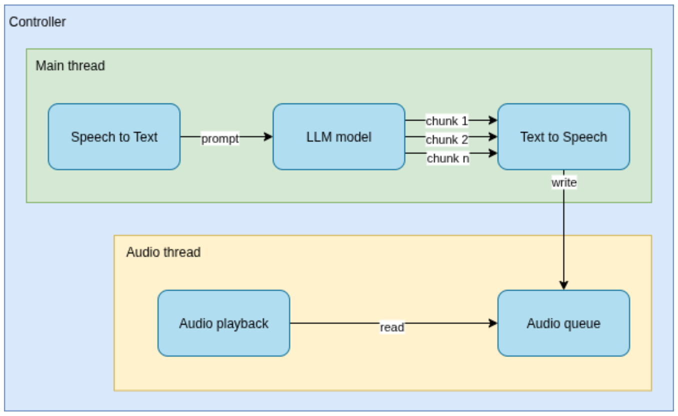

Fig. 2. Python execution flow: microphone listener, STT, LLM streaming, TTS synthesis, and queued playback.

it also coordinates with a synchronization event so that subsequent turns start only after the queue is drained. TTS execution is profiled via the monitoring decorator (label "TTS").

5) Controller and Turn-Taking: The main loop initializes components, starts the audio player thread, and advertises readiness (`assistant/state = ready`). After detecting the wake word on a fresh recording, the controller publishes `assistant/state = listening`, runs the full STT - LLM - TTS turn, and then returns to the ready state. A graceful shutdown (triggered from the Node-RED dashboard) is supported.

E. MQTT Bridge and Remote Control

MQTT is used as the messaging backbone because its lightweight publish/subscribe model is the de-facto standard in IoT deployments and naturally decouples producers (assistant modules) from consumers (dashboards, databases, other services).

MQTT connectivity is established on startup; `assistant/cmd` is subscribed for control messages. A `shutdown` payload triggers a clean stop and emits `assistant/state = stopped`. A convenience `publish` wrapper is used across modules to emit events and state updates.

F. Resource Monitoring

The `monitor_resources` decorator starts a sampling thread that polls process CPU% and RAM usage at fixed intervals while the wrapped function executes, logging average and peak values upon completion. This approach yields per-stage footprints without intrusive instrumentation.

G. Node-RED Orchestration and UI

A Node-RED flow is provided with three tabs: *Installer*, *Execution*, and *Metrics and State*. The *Execution* tab exposes dashboard buttons to start/stop the assistant; the start button executes the Python main script and also emits `assistant/state = started`, while the stop button publishes `assistant/cmd = shutdown`. The *Metrics and State* tab subscribes to `assistant/#`, normalizes messages, and fans out to SQLite and InfluxDB; it also renders a live state (`assistant/state`), the last transcript (`assistant/stt`), and the current utterance (`assistant/llm_chunk`) in the dashboard.

Node-RED is widely adopted in IoT prototyping; here it plays the role of a low-code orchestrator that connects the edge node to storage backends and dashboards without changing the Python runtime.

1) Normalization and Dual-Sink Write: The function node builds a parameterized INSERT into `events(ts, topic, payload)` and prepares an InfluxDB write to measurement `assistant_events`. The Influx payload contains the MQTT topic as a field, and an associated Flux query performs a per-minute `count` by mapping the field to a string and grouping by that mapped value to produce series per topic.

2) SQLite and InfluxDB Configuration: The SQLite node uses a prepared statement for inserts and a dashboard table to inspect recent events. The InfluxDB v2 nodes are configured for a local instance/bucket and a Flux query that counts messages per topic per minute, rendering the result as multi-series area chart in the dashboard.

H. Data Flow and Topics

During a typical turn: `assistant/state` announces readiness and phase transitions; STT publishes the recognized text to `assistant/stt`; the LLM stream emits incremental phrases to `assistant/llm_chunk`; the TTS/audio player sets `assistant/state = speaking`; and finalization returns the system to ready. Remote control is available via `assistant/cmd`.

I. Implementation Notes

- **Chunking policy.** Buffering until punctuation and minimum length optimizes TTS performance and produces natural prosody while maintaining low latency.
- **Queue discipline.** A single FIFO queue and a completion event ensure ordered playback and clean turn boundaries.
- **Fault handling.** MQTT control allows graceful shutdown from the dashboard; exceptions in the audio player do not block the main loop.
- **Observability.** Per-stage resource logs, MQTT event streams, and Node-RED dashboards provide sufficient evidence for performance and reliability analysis without external dependencies.

IV. Results

The assistant and its installer are tested on a *Windows 11 laptop* (Intel i7 CPU, 16 GB RAM), with *Python 3.11* version and using *x64 Native Tools Command Prompt for VS 2022*. It is required to also install Visual C++ Build Tools and CMake. Those dependencies are required only at install time, if the installation step is already terminated successfully, Visual C++ Build Tools and CMake can be uninstalled and the project folder moved on a different device.

A. System Performance

The installation size of the environment, including models and dependencies, is approximately 4.97 GB. Once initialized, the system runs completely offline without requiring external connectivity.

The quantitative evaluation of initialization and processing times for the main components of the assistant is reported in Table I. The results show that all modules are initialized in less than 1.5 seconds, with *Whisper Tiny* requiring the shortest load time (0.095 s) and *SpeechRecognition* the longest (1.256 s) due to ambient noise calibration (1 second). During operation, *Whisper Tiny* achieves transcription in approximately 2.6 seconds per input, averaging different audio input lengths that resulted in 30 character transcriptions; while *Tacotron2 + HiFi-GAN* synthesis averages 3.7 seconds per response, considering on average 73 character length sequences. *TinyLLaMA* leverages a streaming output mechanism, which avoids measurable per-prompt delays and provides near-instantaneous first tokens. Overall, these values confirm the feasibility of real-time interaction: while TTS remains the most time-consuming stage, overlapping playback with synthesis mitigates its impact on user experience.

A complementary view based on aggregated indicators is provided in the following *Metrics* subsection (Section IV B), where telemetry-driven summaries are reported in the added table.

In an IoT context, these measurements define the deployment envelope: the current configuration targets edge nodes with multicore CPUs and a few gigabytes of RAM, while more constrained boards would require model compression or lighter architectures.

The measurements in Table II show that computation is dominated by the STT and TTS stages. *Whisper Tiny (STT)* maintains a high average CPU load of 347.40% with peaks at 487.50% (i.e., roughly 3.5–4.9 logical cores fully utilized), while *Tacotron2-DDC + HiFi-GAN v2 (TTS)* averages 336.98% with peaks at 453.10%. By contrast, *SpeechRecognition* remains lightweight on average (12.04%), with only brief spikes up to 81.20% during capture/calibration. CPU values above 100% are expected in multi-core systems and represent the usage per process summed between cores. Memory usage is comparatively flat across modules (about 1,959–2,151 MB). For *TinyLLaMA (LLM)* only memory was recorded (1959.36 MB), likely because the generation call was not wrapped by the sampler; capturing its CPU profile would require monitoring the streaming generation function. Overall, these results confirm that real-time operation is feasible on a laptop-class multicore CPU, while identifying STT and TTS as the main bottlenecks; on lower-core or embedded targets, mitigation such as thread limiting, lighter vocoders, or tighter chunking would be advisable.

The use of SpeechRecognition ensured that recordings were terminated as soon as silence was detected, lowering the average amount of processed audio per interaction. This mechanism contributed to reduced latency in practical use and minimized unnecessary computation. Whisper Tiny model provided a strong solution for speech-to-text translation, reducing both the disk space usage and the latency translation. The use of TinyLLaMA with the option of streaming the response in small chunks provided a massive boost in responsiveness to the prompt. This also improved efficiency in the text-to-speech translation step with Tacotron2-DDC + HiFi-GAN v2, allowing smaller multiple syntheses instead of processing a single longer text. Implementing the pipeline using these configurations allowed the system to synthesize the next audio files while reproducing previous chunks in the background.

B. IoT Telemetry (InfluxDB) — Results

Telemetry collected via Node-RED and written to InfluxDB summarizes message volumes by topic. Counts are aggregated in one-minute windows and grouped by topic, providing a live view of throughput for `assistant/stt`, `assistant/llm_chunk`. The Node-RED dashboard reproduced the Influx query and displayed per-topic message counts (events/min) over the last hour.

This time-series view of MQTT traffic mirrors common IoT monitoring setups, where message throughput and topic-level activity are key indicators of system health.

C. Qualitative Evaluation

The three stages of the assistant were qualitatively evaluated through example interactions:

- **Speech-to-Text (STT):** The Whisper Tiny model mostly correctly transcribed the prompts, even with a moderately noisy environment. For example, the spoken input *"What are the most used programming languages?"* was correctly transcribed. Misrecognitions occurred with strong background noise and pronunciation inaccuracies.
- **Language Model (LLM):** TinyLLaMA generated coherent and contextually appropriate responses, such as responding with a coherent list of programming languages, such as Python, Java, C# and others. Moreover, it included concise descriptions of them.
- **Text-to-Speech (TTS):** The Tacotron2-DDC + HiFi-GAN v2 pipeline produced natural-sounding voices with smooth intonation. Speech was intelligible and pleasant to listen to, although some prosodic variation was limited compared to commercial systems.

D. Usability Observations

Wake-word detection using Whisper Tiny was reliable and avoided the need for a separate library. The streaming playback significantly improved the user experience, as responses were heard almost immediately without noticeable gaps. The threading mechanism prevented audio overlap or missed inputs.

The integration of `SpeechRecognition` allowed the assistant to adapt to different noise conditions and provided more natural interactions. Users could speak freely without being constrained by fixed-duration recording windows, making the system feel more responsive and intuitive.

The system proved usable in real-time scenarios, suggesting that its design could be deployed in interactive installations or robots where offline operation and privacy are required.

E. Limitations and future work

Although the current prototype demonstrates the feasibility of an offline voice assistant with real-time usability, several directions can further improve its robustness and applicability:

- **Noise robustness:** Future iterations should incorporate techniques such as microphone arrays with beamforming, noise reduction preprocessing, or fine-tuned variants of Whisper to improve recognition accuracy in high-noise environments.
- **Model optimization:** To reduce storage footprint and memory requirements, methods such as quantization (e.g., INT8), pruning, or knowledge distillation could be applied. These optimizations would enable deployment

on more constrained devices, such as Raspberry Pi boards or mobile robots.

- **Multilingual and domain-specific support:** Expanding the assistant to support multiple languages or specialized vocabularies (e.g., healthcare, robotics) is critical to broader adoption in public spaces and professional applications. Transfer learning approaches offer a lightweight path to adaptation.
- **Enhanced interaction modalities:** Integrating additional user interface elements, such as visual feedback (LED indicators or displays), touch interaction, or gesture recognition, could make the system more intuitive and accessible. In the current state, when the assistant is executed, the terminal prints its current elaboration phase.
- **Quantitative evaluation and user studies:** A formal benchmarking campaign using metrics such as Word Error Rate (WER), Mean Opinion Score (MOS) for speech synthesis quality, and latency breakdowns would provide stronger evidence of system performance. Complementary user studies could assess perceived usability and naturalness.
- **Integration of richer metrics and dashboards:** Tracing different metrics and data on InfluxDB and SQLite, in order to produce more detailed dashboards and provide ready to use information about usability and performances.

These directions would not only strengthen the technical performance of the assistant, but would also expand its applicability in real-world scenarios, ranging from domestic environments to interactive public installations.

V. Conclusions

This work presented the design and implementation of a fully offline voice assistant that integrates speech-to-text with Whisper Tiny, conversational inference with TinyLLaMA, and text-to-speech synthesis with Tacotron2-DDC and HiFi-GAN v2. The system operates entirely on local hardware, thereby reducing latency, and remaining functional without network connectivity. Overall, the prototype behaves as a well-instrumented IoT edge service, illustrating how complex ML components can be integrated into standard IoT monitoring and control workflows.

Beyond the core STT–LLM–TTS loop, the assistant was reframed as an *IoT edge node*. Events and metrics are transported over MQTT, orchestrated in Node-RED, durably logged in SQLite (operational facts), and exposed as time-series in InfluxDB (telemetry signals). This extension provides observability and controllability without compromising the offline design: state changes and intermediate artifacts become inspectable, and dashboards support live monitoring and safe remote control.

The quantitative evaluation confirmed that initialization and processing times remain within practical limits on commodity CPUs; streaming generation and chunked synthesis minimize perceived latency even when TTS dominates compute. Telemetry-driven summaries complement these findings with per-topic throughput. Together, these results indicate that real-time interaction is feasible alongside continuous measurement.

Reproducibility and deployability were addressed through an automated installer and a dedicated Node-RED *Installer* tab. The installer provisions the environment, downloads compact models, and initializes local storage; the dashboard encapsulates setup, verification, execution, and shutdown into one-click operations suitable for lab, classroom, or field demonstrations.

In general, the prototype highlights the feasibility of low-latency assistants that are observable and maintainable as IoT edge applications. Although the current system already meets its primary design goals, further improvements in robustness and adaptability remain desirable. Priority items include stronger noise robustness, on-device optimization (quantization/pruning) for embedded targets, multilingual and domain-specific extensions. These directions define a clear path toward scalable, real-world deployments that retain offline guarantees while benefiting from modern observability and control.

References

[1] P. Zhang, G. Zeng, T. Wang, and W. Lu, "Tinyllama: An open-source small language model," 2024.

[2] M. Gupta, M. Haider, and M. Shabbir, "Development of an ai-powered voice assistant: Enhancing speech recognition and user interaction," *International Journal of Scientific Research in Science, Engineering and Technology*, vol. 12, pp. 717–727, 06 2025.

[3] L. Lazzaroni, F. Bellotti, and R. Berta, "An embedded end-to-end voice assistant," *Engineering Applications of Artificial Intelligence*, vol. 136, p. 108998, 2024.

[4] D. Bermuth, A. Poeppel, and W. Reif, "Jaco: An offline running privacy-aware voice assistant," in *2022 17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pp. 618–622, 2022.

TABLE I
INITIALIZATION AND PROCESSING TIMES OF ASSISTANT COMPONENTS

Module / Model	**Initialization Time (s)**	**Avg. Processing Time (s)**	**Notes**
SpeechRecognition	1.256	-	1 second is needed to adjust for ambient noise
Whisper Tiny (STT)	0.095	2.635	Depending on audio length
TinyLLaMA (LLM)	0.462	-	No processing time due to streaming response
Tacotron2-DDC + HiFi-GAN v2 (TTS)	0.839	3.746	Depending on text length

TABLE II
RESOURCE USAGE SUMMARY BY COMPONENT (SINGLE-CALL MEASUREMENTS WITH LIVE SAMPLING)

Component	**CPU Avg (%)**	**CPU Peak (%)**	**RAM Avg (MB)**	**RAM Peak (MB)**
SpeechRecognition	12.04	81.20	2082.59	2086.11
Whisper Tiny (STT)	347.40	487.50	2040.62	2059.63
TinyLLaMA (LLM)	-	-	1959.36	1959.36
Tacotron2-DDC + HiFi-GAN v2 (TTS)	336.98	453.10	2119.06	2150.98

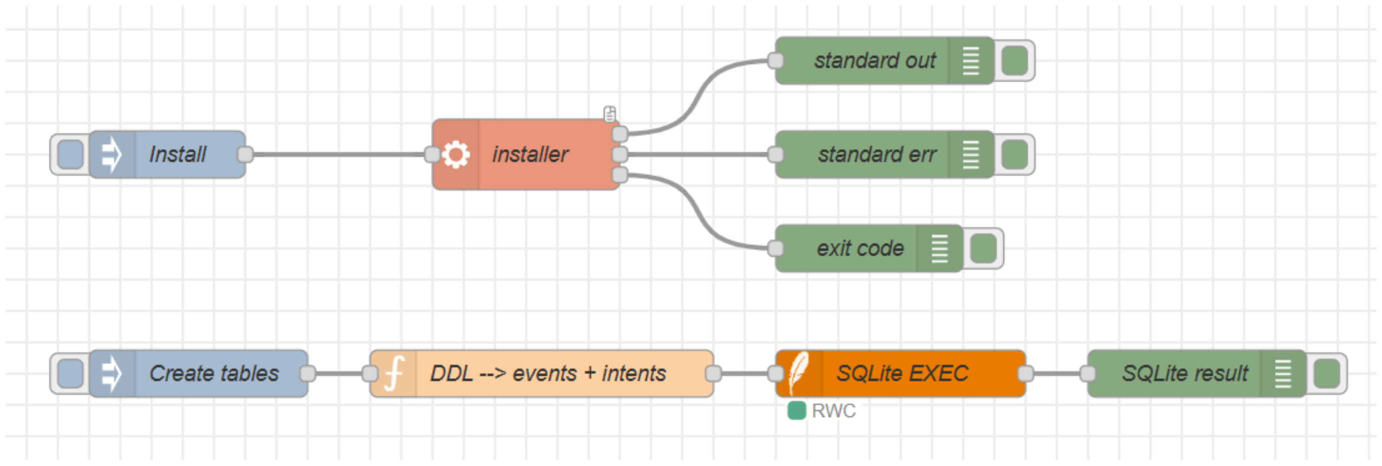

Fig. 3. Installer tab

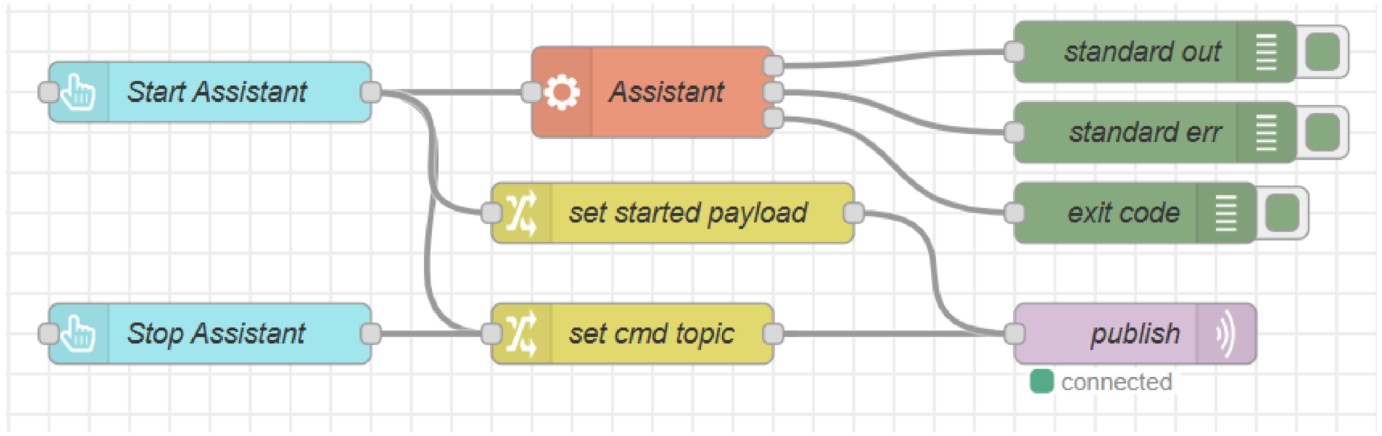

Fig. 4. Execution tab

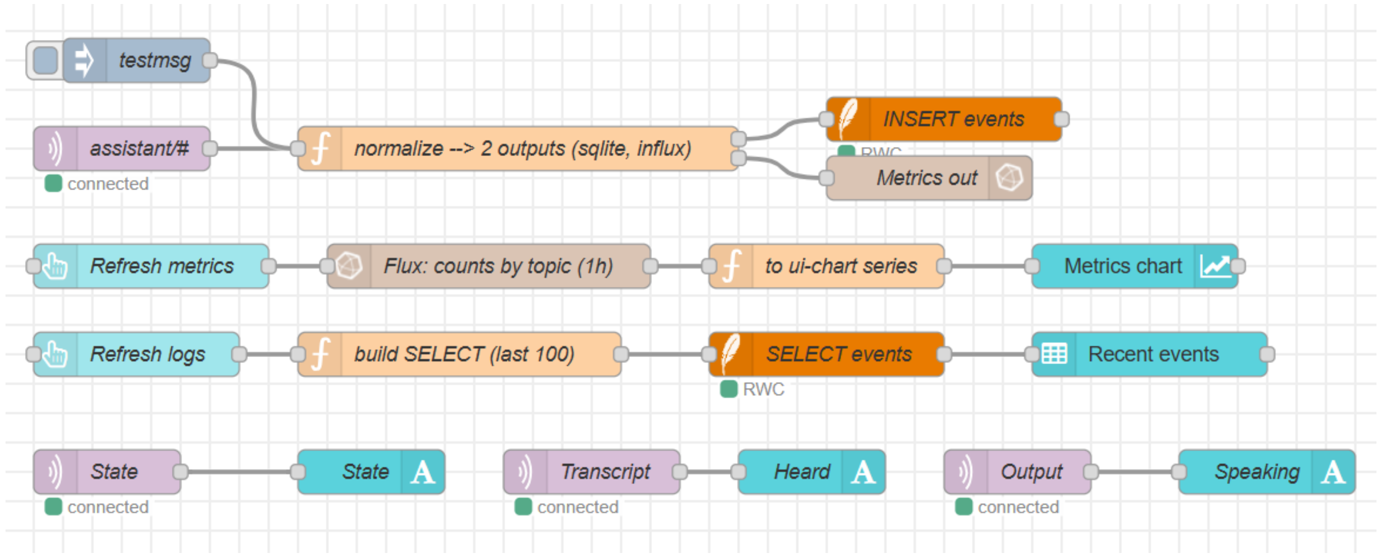

Fig. 5. Metrics & State tab

Fig. 6. Node-RED dashboard flows used for installation, runtime control, and telemetry visualization.