Local Agentic Tool-Calling Speech LLMs

Dylan Lu, Surya Gunukula, Ishan Katpally
Github: https://github.com/ThePickleGawd/realtime-speech-agents

1. Introduction

Speech-to-speech LLMs like GPT-4o have recently gained popularity due to their impressive conversational ability and speed. However, these models are closed-source and rely on expensive, cloud-based infrastructure. To address this, we built a fully local speech-to-speech AI agent that runs entirely on-device. Our system also supports a suite of tools, including our improved implementation of GraphRAG.

2. Methodology

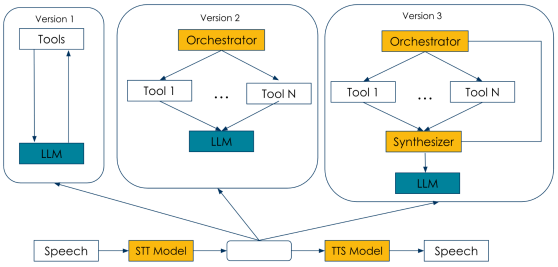


Figure 1: Overview of Model Architecture

The workflow was as follows: user speech is piped into a lightweight speech-to-text model [1], and the entire text output is fed into our tool-augmented LLM. As it generates the response, we stream the output text into a text-to-speech model [2] to get the spoken response. The LLM generation and text-to-speech run on separate threads to minimize latency. After implementing the speech-to-speech framework, we experimented with three different model architectures for the tool-augmented LLM. The first method was an LLM that called tools with JSON and a special tool token, the result of which was then appended to the context, and then continued generating [3]. The next model used an orchestrator LLM which was prompted to select which tools out of the existing tools would be beneficial to call and also format the call for each of them. This would then be parsed to call the appropriate tools, which would again be appended to the context and fed to the LLM to generate an output. The final architecture was similar to the previous, in which an orchestrator LLM would decide what tools to call, but these tool outputs would then be sent a synthesizer model which would be prompted to summarize these tool calls [4]. This would then get passed back to the orchestrator model, so it could decide whether the current added context is enough to answer the query or if additional tool calls must be made. When the orchestrator deems the context appropriate, it is then finally passed on to the LLM. This method allowed the model to answer more complex queries that require multi-hop reasoning, but at the tradeoff of latency. To build these models, we leveraged Langchain's Langgraph framework. This allowed us to develop a modular and flexible multi-agent architecture, where each agent handled a specific task, and it enabled parallel tool execution, and seamless integration with external tools and APIs.

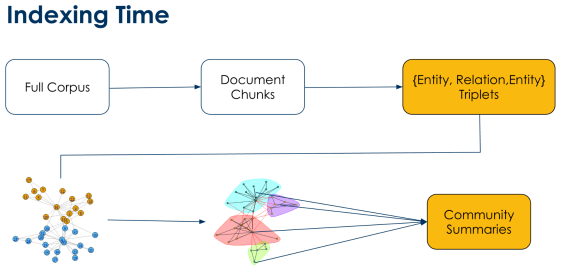
2.1 Tools

The tools that we augmented our LLM with were as follows: DuckDuckGo Web Search, Python Executor, Wikipedia Search, Linux Executor, and GraphRAG. For DuckDuckGo and Wikipedia, we used their Python API library to make the requests. For Python and Linux Executor, we piped the commands or code into a Docker instance and retrieved the output. Finally, our GraphRAG tool was a recreation of Microsoft's GraphRAG [5], but with additional

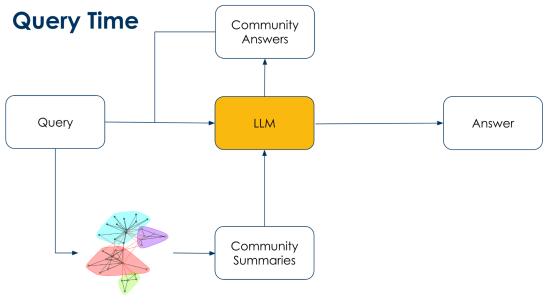
functionality that aimed to improve retrieval accuracy.

2.2 GraphRAG

GraphRAG is split into two components: indexing time and query time. Indexing time is the preprocessing step before the user interacts with the model, and query time is the actions that happen per query.



We adapted Microsoft's GraphRAG pipeline for GitHub repositories. Users submit a GitHub URL, and we parse and chunk the repo files. An LLM extracts {Entity, Relation, Entity} triplets from these chunks to build a knowledge graph. Using Leiden's clustering algorithm, we group the graph into communities and generate summaries for each based on their nodes and associated content.



For a given user query, we augmented Microsoft's GraphRAG methodology to improve retrieval relevancy/accuracy.

At query time, we embed the user query and match it to relevant community summaries in the knowledge graph, which is unlike Microsoft's GraphRAG, which matches to individual nodes. A recent survey [6] found this improves relevance, so we adopted it. For each matched community, we combine its summary and source document chunks as context with the query, prompting the LLM to generate a "community answer." This answer is then passed as the tool call output to our streaming LLM.

2.3 Attention Sinks

While running overnight tests of our agents, we noticed that the LLM's performance breaks down after too many tokens are generated. One possible solution that balances memory efficiency and model performance is to combine sliding window attention with attention sinks [7]. After integrating this cache strategy into our model, we are able to generate tokens infinitely without out-of-memory or high perplexity issues.

3. Experiments/Results

To evaluate our speech models with tool calling, we used an LLM-as-a-judge approach. A recent survey on Agentic RAG showed that this approach is more consistent and stable than professional human evaluation [8]. We curated a dataset of 1000 questions to test each model's answer quality. These questions were designed to use one or more tools to supplement the answer. We fed both the prompt and the model's response into a larger reference LLM (GPT-4), asking it to rate the answer along three dimensions: Clarity, Depth, and Relevance, each on a 0–100 scale. This automated evaluation allowed for fast, consistent benchmarking across different variants of our agentic pipeline. We evaluated the responses from each of the three models, as well as the base model without any tool-calling functionality. We prompted the LLM to provide a brief justification for each score, allowing us to see both quantitative and qualitative differences across versions. We also validated the scores with limited human evaluation and found strong agreement with the justifications, which supported the reliability of this approach.

Prompt: “Who won the Super Bowl this year?”
Base: “The Detroit Lions defeated the Eagles 28–24.”
- Clarity (40), Depth (20), Relevance (10)

V1: “The Eagles won the Super Bowl, beating the Chiefs.”
- Clarity (78), Depth (64), Relevance (78)

V2: “The Philadelphia Eagles won 40–22 vs. the Chiefs. Jalen Hurts was MVP.”
- Clarity (85), Depth (79), Relevance (90)

V3: “The Philadelphia Eagles won Super Bowl LIX on February 9, 2025, with a 40–22 victory over the Chiefs. Jalen Hurts was named MVP after three touchdowns, and the Eagles’ defense intercepted Mahomes twice.”
- Clarity (95), Depth (90), Relevance (96)

Prompt: “When is the next solar eclipse?”
Base: “I’m not sure—my knowledge only goes up to mid-2023.”
- Clarity (50), Depth (10), Relevance (20)

V1: “The next solar eclipse is a partial eclipse on September 21, 2025.”
- Clarity (70), Depth (50), Relevance (70)

V2: “The next solar eclipse is a partial eclipse on September 21, 2025, visible from parts of Australia, the Pacific, the Atlantic, and Antarctica.”
- Clarity (85), Depth (75), Relevance (85)

V3: “The next solar eclipse is a partial eclipse on September 21, 2025. It will be visible across southern Australia, parts of the Pacific and Atlantic Oceans, and Antarctica, with the greatest eclipse occurring around 19:41 UTC. Observers should use certified eclipse glasses for safe viewing.”
- Clarity (95), Depth (88), Relevance (93)

Model	Latency	Clarity	Depth	Relevance
Base	1.6 sec	72	51	75
V1	**2.4 sec**	78	64	78
V2	3.7 sec	**85**	79	**90**
V3	9.4 sec	82	**88**	83

4. Rebuttal

We tested our model with and without the STT and TTS to evaluate its performance. Averaging the results over 100 sample prompts, we saw the latency increase by 0.737s when using TTS and 0.499s for STT. This delay is relatively negligible compared to the LLM's reasoning time. By “tool” use, we refer to two different contexts: 1) Model outputs a special tool token and its desired function in JSON format. On our end, we call the corresponding tool and append the output to the context. 2) Before the LLM starts generating, the Orchestrator outputs a tool call JSON that we parse into relevant tools. The result of these calls gets appended to the context. The novelty of our project is that we tested the accuracy and conversational score of different methodologies of tool-augment generation in its application to Speech LLMs. Furthermore, we improved GraphRAG by editing retrieval to embedding matches to community summaries instead of relevant nodes. Furthermore, we allow an LLM to modify the user query to allow for more accurate matching.

5. References

[1]Beigel, Kolja. Realtime STT. https://github.com/KoljaB/RealtimeSTT

[2]Beigel, Kolja. Realtime TTS. https://github.com/KoljaB/RealtimeTTS

[3]Schick, Timo, et al. "Toolformer: Language models can teach themselves to use tools." Advances in Neural Information Processing Systems 36 (2023): 68539-68551.

[4]Singh, Aditi, et al. "Agentic Retrieval Augmented Generation: A Survey on Agentic RAG." arXiv preprint arXiv:2501.09136 (2025).

[5]Edge, Darren, et al. "From local to global: A graph rag approach to query-focused summarization." arXiv preprint arXiv:2404.16130 (2024).

[6] Zhou, Yingli, et al. "In-depth Analysis of Graph-based RAG in a Unified Framework." arXiv preprint arXiv:2503.04338 (2025).

[7]Xiao, Guangxuan, et al. "Efficient streaming language models with attention sinks." arXiv preprint arXiv:2309.17453 (2023).

[8]Zhu, Weijia, et al. Hallucinations in Tool-augmented Language Models. arXiv, 2024, https://arxiv.org/abs/2411.15594.