SOLA - Smart Offline-first LLM Assistant with Tool Access

E6692.2024Spring.FSCS.report.fps2116.cs4347
Flor Sanders fps2116, Charan Santhirasegaran cs4347
Columbia University

***Abstract*—Voice assistants have become integral parts of modern life, yet concerns persist regarding privacy and security due to reliance on cloud-based processing and data collection. In response, this project introduces SOLA (Smart Offline LLM Assistant), an open-source voice assistant designed for local processing using the Nvidia Jetson Nano Developer Kit. By leveraging a large language model (LLM) at its core and integrating with external tools, SOLA aims to address these concerns while maintaining competitive intelligence. Evaluation results demonstrate SOLA's potential, although challenges remain in optimizing smaller models for effective tool usage. This project serves as a proof-of-concept for local-first voice assistants, highlighting future directions for research and development.**

1. Introduction

We live in a world where technology increasingly permeates our lives. While concepts like ubiquitous computing and the aware home can be dated back to the 1990s, it is only in recent decades that the technology has emerged for their thorough realization [1], [2].

Following in the footsteps of this idea are voice assistants, offered by the likes of Google, Amazon, Apple and Samsung, which provide the user with a natural language interface to a collection of different tools and integrations. However, these services are not without issues or controversy. Due to the fact that a majority of the processing for these assistants occurs in the cloud, they are frequently troubled by security and privacy issues [3], [4]. Moreover, the business model of the companies behind these products rely on massive data collection in a system known as surveillance capitalism [5].

In this project, we address these issues by introducing a fully open source voice assistant. The primary focus of this project is to move as much of the processing required for the operation of the voice assistant from the cloud to the edge as possible. The targeted platform is the Nvidia Jetson Nano Developer Kit. In order to maintain a competitive level of intelligence under these limitations, a large language model (LLM) will be used as the model's core. For tasks requiring access to online services, we provide the assistant to a number of tools that can be implemented as plug-ins. The resulting framework, called SOLA - Smart Offline-first LLM Assistant, is released as open-source software[1].

The remainder of this report is structured as follows. Section 2 provides an overview of related work, including merits and shortcomings. Next, the methodology and implementation of SOLA are discussed in Sections 3 and 4 respectively. Experimental results are discussed in Section 5. Finally, Section 6 and 7 respectively draw conclusions and discuss future work related to this project.

2. Related Work

This section discusses related work. The first section provides and overview of the foremost commercial and open source voice assistants and highlights their merits and shortcomings. Afterwards, past works in LLM tool learning and integration are detailed.

2.1. Voice Assistants

The first modern voice assistant, Siri, was introduced by Apple in 2011. Since then, a number of commercial and open source competitors have emerged, the most popular and relevant of which are listed in Table 1.

Assistant Name	Open Source	Integration Support	Local Processing	Uses LLM
Apple Siri	❌	✅	❌	❌
Google Assistant	❌	✅	❌	❌
Amazon Alexa	❌	✅	❌	❌
Samsung Bixby	❌	✅	❌	❌
Humane AI Pin	❌	❌	❌	✅
Talk-Llama	✅	❌	✅	✅
Mycroft AI	✅	✅	✅	❌
Leon	✅	✅	✅	❌
Rhasspy	✅	✅	✅	❌
SOLA (Ours)	✅	✅	✅	✅

TABLE 1: Overview of voice assistants.

1. https://github.com/FlorSanders/Smart_Offline_LLM_Assistant

Section 1 discussed some of the shortcomings of the voice assistants from the major commercial providers. In addition to the aforementioned points, we note that none of these make use of LLMs and are thus limited in their understanding of the user's natural language inputs. The only exception here is the recently released Humane AI Pin, whose built-in voice assistant proudly makes use of the "latest AI technology". However, their implementation is lacking in support for integrating custom tools.

In recent years, open source voice assistants have largely caught up with the functionality of the commercial options. Being entirely self-hosted, they guarantee privacy to their users. Moreover, Rhasppy and Mycroft specifically target lightweight hardware requirements, allowing processing to occur on single board computer edge devices such as the Raspberry Pi. Like their commercial counterparts, the majority of open source voice assistants rely on intent parsing rather than LLMs for the processing of natural language inputs. The only exception in Table 1 is Talk-Llama, which lacks support for external tool integration.

Comparing with the existing solutions, SOLA is uniquely positioned as an open source LLM-based voice assistant with external tool integration, all while targeting local inference.

2.2. LLM Tool Learning and Integration

As LLMs gained a stronger ability to solve multi-step reasoning tasks and produce a greater range out output formats, they became a prime focus for research into learned tool use. In 2023, Qin et al. introduced ToolLlama, an LLM that could interact with arbitrary APIs to solve complex, multistep tasks [6]. ToolLlama was the result of fine-tuning a Llama model with the Toolbench Dataset, which was created through a procedure using ChatGPT and over 15,000 APIs from various categories on RapidAPI Hub. To build this dataset, GPT-3.5 was used to both generate natural language instructions that would require one or more API calls as well as provide a corresponding solution path through the APIs. They also employed a depth-first search-based decision tree to augment the planning and reasoning of LLMs, since even GPT-3.5 often struggled with more complex multi-step planning and reasoning tasks. Qin et al. also developed an automated evaluator ToolEval to assess models' tool use capabilities by measuring the LLM's ability to accurately follow the commands given (pass rate) as well as tracking the quality and usefulness of different valid solution paths (win rate).

Earlier this year, Guo et al. built upon the aproach presented in the ToolLlama paper to create StableToolBench, which proposed a virtual API server that avoided the problems arising from unstable API statuses [7]. This approach also includes a more deterministic evaluation system utilizing GPT-3.5 for calculating pass and win rates, leading to more stable performance evaluations.

Finally, in their 2024 paper Shen et al. propose an approach that separates the planning, tool use, and result summarization steps and assigns each to a separate LLM fine-tuned on these respective tasks [8]. These LLM "agents" are taught to communicate with each other to process the command through the pipeline and return the desired response. The authors also propose a two part training process in which all three LLMs are first fine-tuned on the overall task and subsequently trained further on their subtask, leading to greater performance than single LLM approaches.

3. Methodology

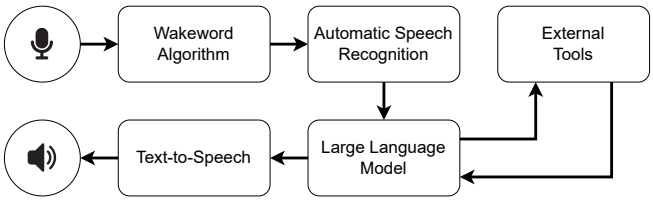

Figure 1: SOLA pipeline system description.

We break down the problem of building a voice assistant by modeling what happens between the microphone input and the speaker output as a pipeline of smaller tasks shown in Figure 1. A description of each step in the pipeline follows below.

1) The microphone input is processed by a lightweight wakeword algorithm, which is triggered whenever an activation phrase is detected.
2) Upon triggering of the wakeword algorithm, a transcription of the user's voice input is generated by means of an automatic speech recognition (ASR) model.
3) The voice transcription is fed to a LLM, which is tasked with intent parsing and answer formulation.
4) If needed, the LLM can make use of a set of external tools to help answer its prompt.
5) Once the LLM has accumulated enough information to answer the prompt, its response is fed to a text-to-speech (TTS) engine.

This system is very similar to the one described in [9], with the main exception being that the intent handling task is performed by a large language model.

4. Implementation

This section discusses the implementation details of SOLA. Specifically, we discuss the software implementation details, the LLM model selection and finetuning process and the details regarding system deployment on the Nividia Jetson Nano.

4.1. Software Design

The implementation for SOLA is largely guided by its system-level pipeline description from Figure 1. The specific choices for our implementation details are motivated by the following principles.

- Modular Design: Each component in the pipeline is implemented as an independent, testable component.
- Model Variety: The implementation of each component supports multiple frameworks and models through the same interface.
- Central Configuration: All configuration parameters are available in a single configuration file.[2]

The resulting software design is a collection of hierarchical building blocks which together implement the pipeline, as shown in Figure 2. Below follows a brief discussion of the implementation details of every component.

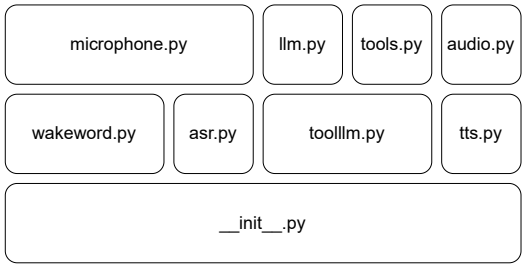

Figure 2: SOLA pipeline software architecture.

4.1.1. Microphone and Audio. The microphone[3] and audio[4] modules simple wrappers around the PyAudio library, which enables interfacing with the host system's audio devices.

4.1.2. Wakeword. The wakeword[5] module implements the wakeword detection algorithm. Its execution relies on the openWakeWord framework, an entirely open source package striking a good balance between performance and computational demands. Additionally, it streamlines the process of training custom models using synthetic data generated via text-to-speech manipulation, augmented by noise, ambient sounds and impulse responses to replicate echo effects. This feature is used within this project to train a custom "Hey, SOLA" wakeword phrase.

4.1.3. Automatic Speech Recognition. The field of speech-to-text engines, also known as automatic speech recognition (ASR), is a constantly evolving research area. While human performance parity has been reached over 5 years ago, the most advanced models unfortunately require a large amount of computing resources, thus hindering their deployment on edge devices [10]. Considering this trade-off between performance and resource usage and the evolving nature of the available models, it is of utmost importance to support multiple frameworks by means of modular software design. In this project's ASR module[6], we integrate the following:

- Vosk: A speech recognition toolkit with very light-weight models available, each only of size 50 MB, capable of running on CPU in real-time.
- Coqui STT: A continued implementation of Mozilla Deepspeech, offering support for various models of different size and capabilities through a simplified API.[7]
- OpenAI Whisper: A well-established transformer model, whose tiny variant is only 150 MB in size and can thus be deployed on the Jetson Nano for local inference.

4.1.4. Large Language Model. The LLM serves as the core of the SOLA voice assistant and poses the project's biggest challenge. With a plethora of commercial and open-source LLM options emerging, the choice significantly affects downstream application performance [11]. Balancing data locality, deployment costs, and computational demands is crucial in selecting an LLM for the voice assistant. Further details on model selection, fine-tuning, and performance are discussed in Section 4.2.

To facilitate switching between different LLM models and hosting environments, the LLM module[8] in this project makes use of a HTTP API interface over which the model is prompted.

4.1.5. Tools. Integration with a variety of tools is what makes a voice assistant useful in practical settings. It allows the system to respond to user-specific questions like calendar entries, as well as providing a way to access internet services such as weather APIs and knowledge bases.

In the tools module[9] of this project, the following three tools are implemented.

- Algebra: Accepts a string with numbers as an input and elementary algebraic operators ($+$, $-$, $*$, $/$) and evaluates the result using Python.
- Weather: Accepts a location as an input (e.g. "New York City") and returns real-time weather information (temperature, relative humidity, precipitation and wind speed) retrieved from the Open-Meteo API.

2. config.json
3. microphone.py
4. audio.py
5. wakeword.py
6. asr.py
7. The company maintaining this library shut down in January of 2024 and henceforth no active maintenance should be expected.
8. llm.py
9. tools.py

- Search: Accepts a keyword search query as an input and returns a Wikipedia description, retrieved from the DBpedia linked open data knowledge graph.

Aside from these tools, one more virtual tool called Answer is included, which should be selected by the LLM to pass its final response to the user. Though they are rudimentary, the successful integration of these tools serve as a proof of concept for tool integration with LLM-based voice assistants.

4.1.6. Tool-LLM. The Tool-LLM module[10] of this project integrates the Tools and LLM modules by recursively prompting the LLM with a selection of tools and the information resulting from their previous calls. The specific details regarding prompt construction and exception handling is discussed in Section **??**.

The prompt template used to enable tool usage in the LLM is provided in Listing 1. It provides the LLM with its identity as a voice assistant, explains the tools it has access to and the expected output format, includes information of previous tool calls and ends with the user's actual prompt.

Listing 1: Tool-LLM prompt template.

```
Your name is Sola, you are a helpful \
voice assistant. Keep responses short.
---
Here are the tools you can use:
- {tool_name}: {tool_description}. \
  (args: {tool_arguments})
You MUST always use one of the tools.
Answers MUST be formatted in JSON format \
with "tool": name-of-tool and "arg": \
arg-value keys.
---
By using the tools you have obtained \
this information:
- Tool "{tool_name}" with argument \
  "{tool_arg}" returned "{tool_result}.
- WARNING: {warning_message}
Once you have enough information, make use \
of the "answer" tool to provide a response \
to the user.
---
{user_prompt}
```

4.1.7. Text-to-Speech Synthesis. As with ASR, text-to-speech (TTS) synthesis is an active field of research that has produced a wide variety of models with different levels of performance and computational requirements. Hence, here too we have opted to integrate multiple implementations in the TTS module[11], all of which can be called over the same interface. Specifically, support is available for:

- Piper: A light-weight neural TTS system developed for the Rhasppy voice assistant.[12]
- Mimic 3: A light-weigth neural TTS system developed for the Mycroft voice assistant.
- Coqui TTS: A continued implementation of Mozilla TTS, offering support for various TTS models of different sizes and capabilities through a simplified API. [13]

4.2. Large Language Model Integration

When planning our approach for developing a tool-using LLM, we decided that due to the context of our project and the accompanying limitations the multi-agent approach discussed in Section 2.2 would be infeasible. Instead, we decided to utilize the ToolBench Dataset to fine-tune a single LLM for integration into the SOLA pipeline. The following sections detail our model selection process along with our fine-tuning and prompt engineering approaches.

4.2.1. Model Selection. When selecting a model for fine-tuning, we first considered the limitations that came with different deployment environment options. The four main options for LLM deployment and their characteristics are described briefly below:

- OpenAI API: High performance, not local, privacy not guaranteed, requires subscription.
- Cloud-hosted LLM: Medium-high performance, not local, privacy guaranteed, expensive hosting expenses.
- Locally hosted LLM: Low-medium performance, local, privacy guaranteed, expensive hardware requirements.
- On-device LLM: Low performance, local, privacy guaranteed, low cost hardware.

As one of our main objectives is to eliminate the privacy and security concerns of using a voice assistant, using the OpenAI API was eliminated as an option. Additionally, we wanted to avoid high recurring costs which eliminated deployment on the cloud as an option due to the required subscription for us. Finally, we considered the cost-performance trade-off between locally hosted and on-device LLMs. The cost of the hardware for hosting an LLM locally could limit access to SOLA and reduce its usefulness, so we decided to further investigate the potential of using an on-device LLM.

The main drawback of hosting the LLM on our Jetson Nano devices was the memory of the Jetson Nano, which greatly constrained the size of the LLM that we could use for tool use. As the Nano only has 2GB of system memory, none of the larger open-source models

10. toolllm.py
11. tts.py

12. Due to compatibility issues, this model has been removed in the Nvidia Jetson Nano deployment codebase.
13. The company maintaining this library shut down in January of 2024 and henceforth no active maintenance should be expected.

would be able to run on-device. Fortunately, we found an open source LLM with only 1.1B parameters, called TinyLlama, which after quantization fits into the Jetson Nano's memory footprint [12]. While using these smaller models solved the memory-related concerns, it was still unclear whether the decrease in model size would lead to a significant sacrifice in the ability to learn tool use.

4.2.2. Model Fine-tuning. In order to fine-tune TinyLlama we used Quantized Low Rank Adaptation (QLoRA), which is a modification of LoRA that allows fine-tuning of quantized models without the need to store the unquantized version in memory for weight updates [13]. LoRA is a fine-tuning approach that calculates the change in weights needed to fit a fine-tuning dataset without permanently modifying the pretrained model weights themselves. It artificially constrains this weight change matrix to a low dimensional subspace by decomposing it into two low rank matrices and only performing weight updates to these matrix factors. By doing this, the number of parameters to tune is greatly reduced while retaining the ability to update all the weights of the original model layer.

We used QLoRA to fine-tune TinyLlama with Unsloth, an accelerator for parameter efficient fine-tuning (PEFT) of LLMs. To do so, we also needed to determine a few hyperparameters including rank, alpha, and target modules. The rank hyperparameter determines the size of the matrix factors that are multiplied to create the final weight update matrix, which influences the dimensionality of the subspace on which the weight updates lie. Thus, the rank determines the maximum complexity of the newly learned behavior and the training time required due to its effect on the number of tunable parameters. Since our preliminary experiments revealed that TinyLlama had trouble correctly generating outputs in the correct JSON format for API calls, we chose a rank of 64, which is near the upper end of commonly used ranks (4, 8, 16, 32, 64, 128) but not so large that it significantly increases training time.

The alpha value, after being divided by the rank, determines the scaling factor for the weight update. Before summing the calculated weight updates with the original model weights during a forward pass, the weight update values are first multiplied with the scaling factor. Through this multiplication, the scaling factor determines how much of an effect the newly learned behavior from the LoRA matrices has on the overall model's behavior. Since we were only able to train on about 4% of the full ToolBench dataset due to time constraints, we needed the fine-tuning to have a disproportionate effect on the model behavior and chose an alpha of 192 to achieve a scaling factor of 3. This value could be lowered to achieve a scaling value closer to one if the full training dataset could be passed through the model, but unfortunately it would take approximately 44 days to do so with the Nvidia T4 GPU we had access to via Google Cloud Platform.

The target module hyperparameter effects which parts of the model are affected by the fine-tuning. Since we wanted to maximize the flexibility of the model in the learning process, we chose to target all projection layers.

4.3. System Deployment

The target deployment platform is the Nvidia Jetson Nano 2GB. In line with its nature as an edge-computing device, it has very limited computational resources available. Additionally, having been released half a decade ago, it lacks the software support required for running many modern Python packages. As such, the Jetson Nano has been the biggest constraint determining design decisions during this project.

Aside from the Jetson Nano, a microphone input and speaker output are required for deployment. For demonstration purposes, we made use of a Monoprice condenser microphone and a generic 10W speaker unit, both of which conveniently make use of USB as a power and data interface and come out of the box with Linux driver support.

In order to deploy the voice pipeline (cfr. Section 4.1), it proved necessary to update the Python runtime on the Jetson Nano from 3.6 to at least 3.7. Unfortunately, out of the box this breaks CUDA support. An attempt was made to reproduce the results from [14], which installs Python 3.9 with CUDA support by building Python and PyTorch from source, but with no success. Hence, all models in the voice pipeline are restricted to running on CPU. Luckily, sufficiently lightweight models are available for each of the modules that real-time performance is still achievable.[14]

For the deployment of the voice pipeline, we recommend LlamaEdge for its low-overhead inference implementation, OpenAI compatible API server and its general ease of use. However, software support for the Jetson Nano again proves lacking. Hence, for deployment on the Nano we turn to llama.cpp which offers similar functionality. While also not officially supported, compilation of the source code for inference on the Jetson Nano with CUDA support is possible.[15]

In the end, the memory envelope of the full system still exceeds the 2GB limitation imposed by the Jetson Nano. The biggest contributor to this is the TinyLlama language model, using over 1GB of RAM. As it stands, the system can either be deployed on a cluster of two networked Jetson Nano devices or on a single 4GB-version of the Jetson Nano, though this last option remains untested.

14. Specific instructions for the deployment of the voice pipeline on the Jetson Nano are available in the repository.

15. Specific instructions for the deployment of llama.cpp on the Jetson Nano are available in the repository.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
GPT 4	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ❌ ✅	✅ ❌ ✅
GPT 3.5	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ❌ ✅	✅ ❌ ✅	✅ ❌ ❌	✅ ❌ ❌	✅ ❌ ❌
Llama 3 8B Instruct (No FT)	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ✅	✅ ✅ ❌	✅ ✅ ✅	✅ ❌ ✅	✅ ❌ ❌	✅ ❌ ✅	✅ ❌ ✅	❌ ❌ ❌	✅ ❌ ❌
Gemma 2B Instruct (No FT)	✅ ✅ ✅	✅ ✅ ❌	✅ ✅ ❌	✅ ✅ ❌	✅ ❌ ❌	✅ ❌ ❌	✅ ❌ ❌	✅ ❌ ❌	✅ ❌ ❌	✅ ❌ ❌	✅ ❌ ❌
TinyLlama (No FT)	❌ ✅ ✅	❌ ✅ ✅	❌ ✅ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌
TinyLlama (FT)	❌ ✅ ✅	❌ ✅ ❌	❌ ✅ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌	❌ ❌ ❌

TABLE 2: Tool-LLM evaluation results.
The results indicate (1) if the model produced correct JSON outputs,
(2) whether the correct number of tools was used and (3) if the final answer is correct.
✅ represents a passing result, while ❌ means failing.

5. Results

In order to evaluate the performance of the LLMs as a tool-using voice assistant, we create a set of eleven benchmark questions with increasing difficulty as the number of tools, language understanding and creativity required to solve the prompt increases.

1) What is your name?
2) What can you help me with?
3) Write me a poem about the tools you have access to.
4) What is the current weather in Columbia University NYC?
5) What is fifty seven multiplied by three hundred and twenty one minus four?
6) In what city did the 2020 olympic games take place?
7) What is the current weather in the city where the 2020 olympic games took place?
8) What is the sum of the current temperature and humidity at Columbia University?
9) What is the product of all the numbers of the year when the Olympic Games were held in Beijing?
10) Write a poem about the current weather in the city where the 2020 olympic games took place.
11) What is the sum of the current temperature and humidity in the city where the 2020 olympic games took place?

We feed these prompts into a number of different LLMs of various sizes, including GPT 4 and GPT 3.5 as well as the recently released open source Llama 3 and Gemma models and finally Tinyllama before and after fine-tuning. For each model and prompt, we record the following properties.

1) Does the model produce an output in valid JSON format, as instructed by the prompt of Listing 1?
2) Does the model use the expected number of tools required to solve the question?
3) Is the answer provided both accurate and satisfactory?

The results of these experiments are displayed in Table 2, with the models sorted by decreasing number of parameters and the questions by increasing difficulty. The name of each model in the table links to the log files, which are published in our repository for future reference.

We see that the GPT 4 benchmark produces a satisfactory result for all of our questions, even though for Q10 and Q11 it relied on its general knowledge to know where the 2020 olympics took place, rather than confirming it with the search tool. As the number of parameters decreases from this reference point, the models start having trouble with more and more of the complex questions. For Gemma, web observe that it is able to produce correct JSON outputs without a problem, but then consistently lacks the reasoning power to use the gathered information to formulate a response. In case multiple tools need to be used, this model gets confused and fails entirely.

The TinyLlama model without fine-tuning fails to produce any valid JSON output at all and as such is unable to access any of our tools. The reason for this is that TinyLlama has been trained with a data sampling strategy that prioritizes natural language prompts over coding prompts by a 7:3 ratio [12]. Considering that only a limited number of coding prompts will include JSON data structures, the model's struggle is understandable. The goal of finetuning this model on the Toolbench dataset was to familiarize the model better with JSON data structures, such that it would be capable of producing valid prompt outputs. Unfortunately, the results show that our efforts were in vain as the model is still unable to produce outputs in the desired format and even lost some of its natural language reasoning capabilities. We hypothesize three reasons for this results:

- Training the model on only a fraction of the Toolbench data set might not have given TinyLlama enough opportunity to learn what JSON data is supposed to look like.
- The prompt format used in Toolbench is somewhat different from the one adopted in this project. Adapting these prompts to be more similar may help TinyLlama perform better on this task.
- The model architecture of TinyLlama may just be too limited to learn both a good level of natural language understanding and the capability to produce correct JSON API calls.

This last point is supported by prior research, which

has shown that smaller LLMs often struggle with performing the different tasks required for tool learning [8].

6. Conclusion

In conclusion, SOLA offers a promising solution to privacy concerns in voice assistant technology. By leveraging open-source tools and local processing, it prioritizes user privacy while maintaining functionality. The methodology outlined in this report provides a roadmap for developing similar solutions, emphasizing modular design and flexibility. However, challenges remain in optimizing smaller models like TinyLlama for effective tool usage. With further research and refinement, solutions like SOLA have the potential to reshape voice computing, prioritizing privacy without sacrificing usability.

7. Future Work

This project serves as a proof-of-concept implementation of a fully open source and local-first voice assistant with LLMs at its core. Comparing to the state of the art in both commercial and open source voice assistants, this is a unique combination. On the other hand, our implementation lacks a number of features that are offered by the competition, that are in further stages of development.

- Support for multiple languages.
- Integration of significantly more tools.
- Multi-modal input support.
- Self-contained docker deployment.

Additionally, further research is needed to successfully make use of smaller LLMs for tool learning applications like voice assistants. The development of small LLMs is luckily ever ongoing, with several new models having been published throughout the development of this project [15], [16]. Such innovations, along with a more thorough investigation of model finetuning for tool usage will lead to improvements in the performance and usability of local LLMs for voice assistant applications.

8. References

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 66–75, Jan. 1991. [Online]. Available: http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html.

[2] C. D. Kidd, R. Orr, G. D. Abowd, *et al.*, "The aware home: A living laboratory for ubiquitous computing research," in *Cooperative Buildings. Integrating Information, Organizations, and Architecture*, N. A. Streitz, J. Siegel, V. Hartkopf, and S. Konomi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 191–198, ISBN: 978-3-540-48106-5.

[3] P. Cheng and U. Roedig, "Personal voice assistant security and privacy—a survey," *Proceedings of the IEEE*, vol. 110, no. 4, pp. 476–507, 2022. DOI: 10.1109/JPROC.2022.3153167.

[4] G. Germanos, D. Kavallieros, N. Kolokotronis, and N. Georgiou, "Privacy issues in voice assistant ecosystems," in *2020 IEEE World Congress on Services (SERVICES)*, 2020, pp. 205–212. DOI: 10.1109/SERVICES48979.2020.00050.

[5] S. Zuboff, *The Age of Surveillance Capitalism: The Fight for a Human Future at the New Frontier of Power*. PublicAffairs, 2019, ISBN: 9781610399050. [Online]. Available: https://books.google.com/books?id=u3GmwgEACAAJ.

[6] Y. Qin, S. Liang, Y. Ye, *et al.*, *Toolllm: Facilitating large language models to master 16000+ real-world apis*, 2023. arXiv: 2307.16789 [cs.AI].

[7] Z. Guo, S. Cheng, H. Wang, *et al.*, *Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models*, 2024. arXiv: 2403.07714 [cs.CL].

[8] W. Shen, C. Li, H. Chen, *et al.*, *Small llms are weak tool learners: A multi-llm agent*, 2024. arXiv: 2401.07324 [cs.AI].

[9] M. Hansen, J. Thalheim, and N. Bento, *Rhasspy*, https://github.com/rhasspy/rhasspy3, 2024.

[10] W. Xiong, L. Wu, F. Alleva, J. Droppo, X. Huang, and A. Stolcke, "The microsoft 2017 conversational speech recognition system," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, Apr. 2018. DOI: 10.1109/icassp.2018.8461870. [Online]. Available: http://dx.doi.org/10.1109/ICASSP.2018.8461870.

[11] W. X. Zhao, K. Zhou, J. Li, *et al.*, *A survey of large language models*, 2023. arXiv: 2303.18223 [cs.CL].

[12] P. Zhang, G. Zeng, T. Wang, and W. Lu, *Tinyllama: An open-source small language model*, 2024. arXiv: 2401.02385 [cs.CL].

[13] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, *Qlora: Efficient finetuning of quantized llms*, 2023. arXiv: 2305.14314 [cs.LG].

[14] X. Geerinck, *Enabling cuda with pytorch on nvidia jetson and python 3.9*, https://xaviergeerinck.com/2021/11/25/enabling-cuda-with-pytorch-on-nvidia-jetson-and-python-3-9/, 2021.

[15] S. Mehta, M. H. Sekhavat, Q. Cao, *et al.*, *Openelm: An efficient language model family with open training and inference framework*, 2024. arXiv: 2404.14619 [cs.CL].

[16] M. Abdin, S. A. Jacobs, A. A. Awan, *et al.*, *Phi-3 technical report: A highly capable language model locally on your phone*, 2024. arXiv: 2404.14219 [cs.CL].

9. Appendix

9.1. Individual Student Contributions

TABLE 3: Student Contributions

UNI	fps2116	cs4347
Last name	Sanders	Santhirasegaran
Contribution fraction	60 %	40 %
What I did 1	Implement the voice processing pipeline.	Implement LLM fine-tuning.
What I did 2	Perform system deployment on the Nvidia Jetson Nano.	Optimize training efficiency, hyperparameter tuning.
What I did 3	Perform evaluation of the Tool-LLMs.	Contribute to the report and slides.
What I did 4	Contribute to the report and slides.	