SHANKS: SIMULTANEOUS HEARING AND THINKING FOR SPOKEN LANGUAGE MODELS

Cheng-Han Chiang[1,2*] **Xiaofei Wang**[2†] **Linjie Li**[2] **Chung-Ching Lin**[2] **Kevin Lin**[2]
Shujie Liu[2] **Zhendong Wang**[2] **Zhengyuan Yang**[2] **Hung-yi Lee**[1] **Lijuan Wang**[2]
[1]National Taiwan University [2]Microsoft

ABSTRACT

Current large language models (LLMs) and spoken language models (SLMs) begin thinking and taking actions only *after* the user has finished their turn. This disables the model from interacting with the user during the user's turn and can lead to a high response latency for waiting for the model to think. Consequently, thinking *after* receiving the full input is not suitable for speech-to-speech interaction, where real-time and low-latency interaction is important. We address the above issue by drawing inspiration from the fact that humans can naturally *"think while listening"*. In this paper, we propose **SHANKS**, a general inference framework that enables SLMs to generate unspoken chain-of-thought reasoning when listening to the user input. SHANKS streams the input speech in fixed-duration chunks and, as soon as a chunk is received, generates unspoken reasoning based on all previous speech and reasoning, while the user continues speaking. SHANKS uses unspoken reasoning to determine whether to interrupt the user and make tool calls to complete the task. We demonstrate that SHANKS enhances the real-time user-SLM interaction in two scenarios: (1) When the user is presenting their step-by-step solution to a math problem, SHANKS can listen to and reason over the user's speech and make an interruption when the user makes a mistake. SHANKS interrupts the user 37.1% more accurately compared with a baseline that interrupts the user without thinking. (2) In a tool-augmented dialogue scenario, where the model needs to make tool calls to achieve the user's request, SHANKS can complete 56.9% of the tool calls before the user even ends their turn. Overall, SHANKS is a step toward models that keep thinking throughout the conversation, not only after a turn ends. Animated illustrations of SHANKS can be found at `https://d223302.github.io/SHANKS/`.

1 INTRODUCTION

In recent years, the *thinking* process has been used to improve Large Language Models (LLMs), where the LLM first generates a *hidden* chain-of-thought (CoT) reasoning (Wei et al., 2022; Kojima et al., 2022) invisible to the users, and then generates the final output response (OpenAI, 2024c; Guo et al., 2025). This thinking process improves LLMs on reasoning-intensive tasks, including mathematics (Lightman et al., 2024), coding (Chen et al., 2021), and questions that involve significant domain knowledge (Rein et al., 2024). However, current reasoning LLMs only start to think *after* receiving the complete user input, which is reasonable for turn-based interactions, i.e., the model processes the user's message after it is fully composed and sent.

In contrast, human behavior in *spoken* communication is different. Humans naturally think *while* listening, far before the speaker finishes their turn (Bögels et al., 2015; Corps et al., 2018; Bögels et al., 2018). Thinking during listening offers two key advantages: (1) It enables timely and well-founded reactions, including interruption, even before the speaker's turn ends. (2) It reduces response latency by allowing answer preparation to begin before the speaker finishes speaking. Motivated by these

[*]Work done during an internship at Microsoft GenAI. `dcml0714@gmail.com`
[†]Correspondence: `xiaofei.wang@microsoft.com`

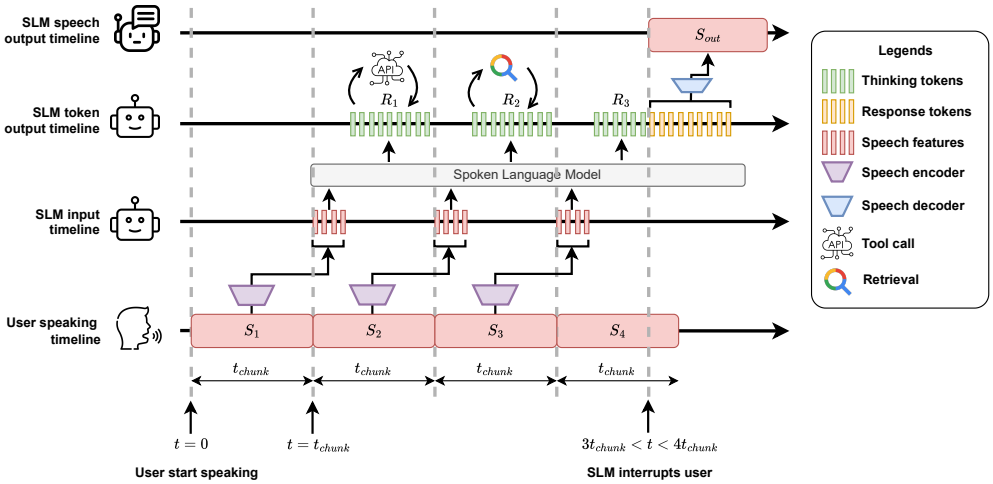

Figure 1: The timing diagram of SHANKS. As the user speaks, their speech is segmented into chunks for every t_{chunk} seconds and streamed to the SLM. After receiving an input chunk, SHANKS generates the thinking tokens, which might include calling external tools or determining to interrupt the user. When the user is speaking the i-th speech chunk S_i, SHANKS generates the $(i-1)$-th thinking chunk R_{i-1}, achieving thinking while listening. When the current speech chunk S_i is fully spoken by the user, SHANKS stops the thinking for R_{i-1}, adds the latest speech S_i and the previous reasoning R_{i-1} to its context, and begins the i-th thinking chunk R_i.

observations, we propose a method to enable spoken language models (SLMs) to think while listening to input speech.

In this paper, we introduce SHANKS: **S**imultaneous **H**earing **a**nd Thi**nk**ing with Chunked Input **S**peech. SHANKS is a general inference framework for both end-to-end (E2E) and cascade SLMs to achieve thinking while listening. At inference time, SHANKS processes the user input in a fixed-size chunk. Once a chunk of speech input is received, SHANKS generates a chunk of unspoken thinking tokens based on all previous input speech chunks and previous thinking chunks. SHANKS alternates between receiving the input speech chunk and generating an unspoken CoT reasoning when the user is still emitting the next speech chunk, achieving the ***thinking while listening***. During the thinking process, SHANKS can decide to interrupt the user or make tool calls to prepare for the final spoken response. The inference workflow of SHANKS is depicted in Figure 1.

We use two scenarios to show how SHANKS can improve real-time user-SLM interaction. First, we study a scenario where the user first describes a math question and then describes their step-by-step solution. SHANKS can listen to the user's problem-solving process and perform internal thinking in the meantime to interrupt the user when the user makes a mistake in their solution. This scenario has great potential in educational use cases, where the SLM serves as a tutor to guide the student. Compared to a baseline that makes an interruption without thinking, SHANKS interrupts 71% more when the user makes a mistake, while the interruption made by SHANKS is 37.1% more valid.

Next, we focus on a task-oriented dialogue setting, where the user requests the SLMs to assist with a travel plan, and the model needs to call Booking.com APIs to complete the user's request and respond to the user. Without SHANKS, all the tool calls can only be made after the user's speech ends, resulting in a higher response latency. We use SHANKS to make tool calls when the user is still speaking. SHANKS enables the model to successfully complete 56.9% of API calls while the user is still speaking, reducing the latency of the final response.

We summarize our contribution as follows:

1. We propose SHANKS, a general framework for SLMs that enables *thinking while listening*. To the best of our knowledge, we are the first to explore generating unspoken CoT reasoning when the user is still speaking.

2. We show that SHANKS can interrupt the user more accurately compared to a baseline that interrupts without thinking.
3. Using SHANKS, the SLM can successfully make tool calls *before the user even finishes talking*.

2 METHOD: SIMULTANEOUS HEARING AND THINKING WITH CHUNKED INPUT SPEECH

Current LLMs and SLMs only start to think after the user's input is completed. In contrast, humans can think while listening (Bögels et al., 2015; Corps et al., 2018; Bögels et al., 2018), where we reason over what we just heard, guess what the speaker might be up to, and prepare the necessary ingredients to cook up a good response. Thinking while listening allows us to interact with the speaker better when the speaker is still speaking. In this section, we introduce SHANKS, a general framework to make SLMs capable of thinking while listening. Here, we only discuss the basic form of SHANKS, and we defer the more advanced usages, including interruption or tool call, to later sections.

2.1 INFERENCE

During inference, SHANKS requires that the user's input speech comes in a streaming manner. SHANKS processes the streaming user input speech by a fixed chunk size of t_{chunk} seconds. We use S_i to denote the i-th user input speech chunk, where S_i is an audio chunk of t_{chunk} seconds, except for the last chunk S_N, which may be shorter. When the user is still speaking, SHANKS alternately takes the user speech S_i and generates the thinking chunks R_i conditioning on all previous user speech and all previous thinking chunks. The thinking chunks R_i are **not spoken by the SLM**; they only serve as the internal reasoning process of the SLM.

Here, we walk through what happens for SHANKS during inference. The following contents are best read with Figure 1. At $t = 0$, the user begins to talk. When $t = t_{\text{chunk}}$, the user speech from 0 to t_{chunk}, i.e., S_1, is sent to the SLM. Here, we append a special token `[EOPA]` (end of partial audio) after S_1 to let the SLM know that this is the end of a chunk of *partial* user speech. Based on S_1, the SLM generates the first thinking chunk R_1. A thinking chunk is enclosed in two special tokens `<think>` and `</think>`. The SLM generates R_1 during the interval $t = t_{\text{chunk}}$ to $t = 2t_{\text{chunk}}$, and the user is still speaking the second chunk S_2 at the same time.

Since t_{chunk} is the time for the SLM to generate its thinking, the duration of t_{chunk} cannot be selected too small; otherwise, the SLM may not be able to produce meaningful thinking chunks. The number of thinking tokens in R_i is restricted to no more than $t_{\text{chunk}} \times n_{\text{tps}}$, where n_{tps} is the number of tokens the model can generate per second. Unless specified, we select $t_{\text{chunk}} = 4.0s$ in our paper; a 7B model can generate around 320 tokens on a single A100 GPU in this duration. At the end of t_{chunk}, if the thinking chunk has not finished generating, i.e., the `</think>` token has not been emitted, we directly stop generating and append the `</think>` token at the end of the thinking chunk.

At $t = 2t_{\text{chunk}}$, we take the freshly obtained user speech chunk S_2 (from $t = t_{\text{chunk}}$ to $t = 2t_{\text{chunk}}$) and pass this chunk to the SLM, and again appending the `[EOPA]` after this chunk to generate the next thinking chunk. (Assume that the user still has not ended their turn at $t = 2t_{\text{chunk}}$.) When generating the next thinking chunk R_2, the SLM conditions on S_1, R_1, and S_2. The SLM will continue the process of taking user input speech chunks and generating the thinking chunks until the user ends their speech in the N-th chunk of speech, S_N. After the user ends their speech, we feed the last speech chunk S_N into the SLM, while this time we append a different special token `[EOA]` (end of audio), indicating that the user's speech has ended. Based on S_N and all the previous interleaved speech/thinking chunks $\{S_1, R_1, \cdots, S_{N-1}, R_{N-1}\}$, the SLM generates the thinking chunk R_N and then generates a final response chunk O. Only the final response O will be spoken by the SLM.

Since SHANKS chunks the user input using a fixed-duration chunk t_{chunk}, the model's thinking will lag behind the user's speech by at least t_{chunk} seconds. If the user's speech is less than t_{chunk}, SHANKS cannot think while listening. However, since long speech can easily happen in real-world interaction, this limitation might not be a significant weakness of SHANKS.

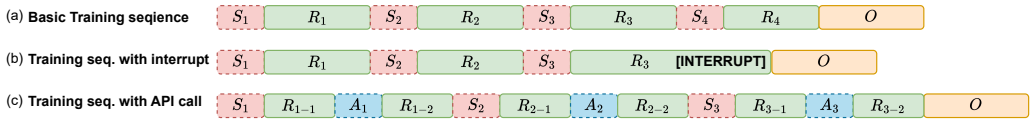

Figure 2: Illustration of the training data. S_i: the speech for the i-th user speech chunk; R_i: the i-th thinking block after S_i; O: the final response block; A_i: the API call responses after the speech chunk S_i. Blocks in dashed lines do not contribute to the training loss, while blocks in solid lines are included for loss calculation. (a) The general training sequence: Alternating between user speech block and SLM thinking token chunks (Section 2.2), followed by a final response chunk. (b) Training data with interruption: Alternating between user speech blocks and the thinking token chunks, while the last thinking chunk includes a special token `[INTERRUPT]`. (c) Training data with API calls: Similar to (a), while each thinking chunk may be separated into two blocks R_{i-1} and R_{i-2} by the API call response A_i.

2.2 Training

During inference, SHANKS requires the SLM to generate thinking chunks based on all previous user input speech chunks and the model's own thinking chunks. During training, we prepare datasets to make the SLM learn this behavior. Assume that we have a complete user speech S, we can segment it into N chunks $\{S_1, \cdots, S_N\}$ with a fixed duration t_{chunk}. Next, assume that we use some method to obtain the thinking chunks $\{R_1, \cdots, R_N\}$ and the output response O; we will explain how to obtain them in later sections. After preparing the training data, we use the standard language modeling cross-entropy loss to train the SLM to predict R_1 given $\{S_1\}$, predict R_2 given $\{S_1, R_1, S_2\}$, etc., and predict R_N and O given $\{S_1, R_1, \cdots, S_{N-1}\}$. A full training sequence is depicted in Figure 2(a).

3 Task Introduction

After introducing the basics of SHANKS, we use two tasks to demonstrate how SHANKS can be applied. In this section, we explain the setup of the two scenarios and how SHANKS works at inference time.

3.1 Scenario 1: Interrupting User Turn

In the first scenario, we aim to use SHANKS to make SLMs able to interrupt the user when the user is saying something wrong. The significance of this application lies in its potential in educational use cases, where the SLMs can serve as a tutor and listen to the speaker, a student, describing how they solve a problem. The SLM can make a timely interruption to let the student know that they are making a mistake, allowing them to correct it as early as possible.

Interruption is related to the full-duplex ability of spoken language models (Lin et al., 2025). While most prior works on full-duplex SLMs focus on user interrupting SLMs, we focus on the reverse scenario: SLM interrupting users. As an important note, we do not advocate that it is good to have a model that interrupts the user. Some users might find it annoying and unpleasant when interrupted by an SLM. Our goal is a modeling mechanism that enables interruption, leaving model deployer policies and users to decide when (or whether) to turn it on.

3.1.1 Task Description

We explain the precise task we are studying. In this task, the user describes a math problem and then solves the problem. The user's solution does not simply state the answer; the user describes a step-by-step problem-solving process, which might be correct or wrong. The SLM needs to interrupt the user when the user is making a mistake, and not to interrupt the user when the solution is correct.

As this is a novel task and there is no available data, we built the evaluation data ourselves. First, we construct the user speech. A single user speech includes (1) a math question and (2) a step-by-step solution. We source the math questions from the testing data of GSM8K (Cobbe et al., 2021), a grade-school math word problem dataset commonly used for evaluating mathematical reasoning

ability (Wei et al., 2022; Kojima et al., 2022; Wang et al., 2023). Next, we use two LLMs, Llama-2-7B (Touvron et al., 2023) and Llama3.1-8B (Grattafiori et al., 2024), to generate step-by-step answers for those questions, and use GPT-4o (OpenAI, 2024b) to determine if the answer generated by the two models matches the ground truth answer in the dataset. We select these two models since they can generate CoT reasoning to solve the math problem, and their performance on GSM8K is very different: Llama-2-7B is a weaker model and prone to generating wrong solutions, while Llama-3.1-8B is a stronger model, which can generate more accurate solutions.

After we have the texts for the step-by-step solution, we convert them into speech. We use GPT-4o to rewrite the answers generated by the two Llama models to make the solution more colloquial. Next, we concatenate the original question, the colloquial step-by-step answer, and prepend a prefix *"I want to solve the following question."* to form the transcription of a testing instance. We use GPT-4o-mini-TTS (OpenAI, 2024a) to synthesize the speech.

The final testing dataset includes 1280 instances with correct solutions and 1140 with incorrect solutions. We call the former the "*correct subset*" and the latter the *"incorrect subset*. The average duration of the user's speech is around 49.25 seconds.

3.1.2 Training Data for Interruption

To teach the model to think while listening and determine whether to interrupt, the training data in this task include two types of instances: (1) The user provides a correct step-by-step solution to the question, and the model does not interrupt the user during the user's speech. After the user finishes the speech, the output response acknowledges the correctness of the answer. (2) The user's turn unfolds an erroneous problem-solving process, and the model interrupts the user when the user makes the first mistake and clearly explains what is wrong.

To construct such a training dataset, we use the math questions in Tulu3-Persona-Math-Grade (Lambert et al., 2024) to construct the user speech S (including the question and the step-by-step solution) following the previously described procedure, and then segment the speech by a fixed duration $t_{\text{chunk}} = 4$ seconds to obtain $\{S_1, \cdots, S_N\}$.

Next, we use GPT-4o to generate the thinking chunk R_i. When generating the i-th thinking chunk R_i, the input to GPT-4o includes the transcriptions of all previous user speech chunks $\{S_1, \cdots, S_i\}$ and all previous thinking chunks $\{R_1, \cdots, R_{i-1}\}$. GPT-4o is required to do the following in the thinking chunk: (1) Track the information already known and calculate intermediate variables when they are available. (2) Identify if any errors are in the current user's transcription. If there is an error, GPT-4o should generate a `[INTERRUPT]` token at the end of the thinking chunk, indicating that the user should be interrupted. We give GPT-4o four in-context examples to allow GPT-4o to understand the task. The prompt to GPT-4o is given in Table 3 and 4 in the Appendix.

After generating the thinking chunks, we generate the final output response O. For the user speeches with an error-free solution, the output response simply needs to let the user know that their solution is correct. We prompt GPT-4o to generate the final response based on the transcription of the full user speech and all previous thinking chunks. Now, we can form a training data sequence by interleaving S_i and R_i and then appending O in the end.

For those user speeches with a wrong solution, the output response will be an interruption to the user's speech. Assume that based on our previous process for generating the R_i's, GPT-4o decides to interrupt the user after the user speech chunk S_k, i.e., the thinking chunk R_k includes the interruption token `[INTERRUPT]`. To generate a response for interruption, we give GPT-4o the user's speech up to the k-th user speech chunk and all the previous thinking, and ask GPT-4o to generate a response O to interrupt the user. The interruption should be precise on what error is made by the user and how to correct it. After this process, we can interleave S_1 to S_k with R_1 to R_k and append O in the end to form a training sequence. A figurative illustration of this training instance is shown in Figure 2(b). Note that in the last thinking chunk R_k, there will be a special token `[INTERRUPT]`, indicating that the model is going to interrupt the user.

3.1.3 Inference and Evaluation

During inference, we stream the user speech to the SLM by a fixed chunk size t_{chunk}, and follow the inference procedure elaborated in Section 2.1. If the SLM generates the special token `[INTERRUPT]`

in a thinking chunk R_k and outputs a response chunk O (when the user is emitting speech chunk S_{k+1}), we convert the response token into speech to interrupt the user. Once an interruption happens, the future user speech chunks will not be streamed to the SLM.

We evaluate a model on the testing dataset constructed in Section 3.1.1. We use the following metrics:

1. **Interrupt ratio**: The ratio of total interrupted instances among the total instances. A good model should have a low interrupt ratio on the correct subset and a high interrupt ratio on the wrong subset.
2. **Valid interrupt ratio**: This is used to evaluate whether an interruption made by the model is valid, and the valid interrupt ratio is defined as the ratio of *valid* interruptions among the total interruptions. To judge whether an interruption response O is valid or not, we use LLM-as-a-judge (Chiang & Lee, 2023; Zheng et al., 2023). We give the judge LLM (GPT-4o) the transcription of the user input until the time of interruption[1] and output response O from the model, and ask the LLM judge to determine if the model's interruption response O correctly interrupts the user when there are unclear or mistakes in the user's speech. The prompt given to the judge model is shown in Table 10 in the Appendix.
3. **Interruption latency**: The time of the model interruption compared to *the time when the first error happens in the user's speech*, denoted as t_{error}. This metric is only used to evaluate the incorrect subset. For samples in the incorrect subset, we use GPT-4o to determine t_{error}. The details on determining t_{error} are included in Appendix C.1. Assume that the model interrupts the user at $t_{\text{interrupt}}$, then the interruption latency is calculated as $t_{\text{interrupt}} - t_{\text{error}}$, where $t_{\text{interrupt}}$ is the time when the model starts to emit the first package of the speech for the interruption output O.

3.2 Scenario 2: Making Tool Calls When Listening

In the previous scenario, we have introduced SHANKS can be used to think when listening to interrupt the user when necessary. However, the "*thinking*" of a model can not only include the CoT reasoning generated by the model itself; the model can use external tools in their thinking process to complete the user's request and improve the answer quality (Schick et al., 2023; Qin et al., 2024b;a; Gao et al., 2024; Wang et al., 2024). In the second scenario, the SLMs will use ***external tools*** in their thinking process to achieve the user's request.

This kind of ***tool-augmented generation*** has been widely explored in text-in-text-out LLMs (Qin et al., 2024a). Given a user request, a tool-augmented LLM selects relevant tools such as calculators (Schick et al., 2023), search engines (Luo et al., 2023), and other APIs (Qin et al., 2024b), and integrates the tool execution results into its reasoning process. However, current tool-augmented LLMs begin calling the tools after the full user input is received, which adds delay while the model invokes tools, waits for results, parses them, and composes a response. This delay may be acceptable in text-only settings, but in spoken dialogue, this latency breaks conversational flow.

In the second task, we will use SHANKS to make tool calls when the user is still speaking, thus reducing the response latency due to making tool calls. As a proof of concept, we consider a task-oriented dialogue where the SLM serves as a travel-agency agent and is given a set of APIs it may call to complete the task. For example, the user might say, *"Help me check the details of the cheapest flight from Hangzhou to Seoul on December 10, 2024, and the car rental information near Seoul airport.*" The SLM makes API calls to resolve airport names or codes, search for flights, and then query car-rental options, and finally composes the results into a single reply.

Given a user query like the example above, once the destination and date are clear, the flight search API can be called even when the user is only halfway through speaking. This is where SHANKS can be useful: processing partial user input and performing early actions. Next, we formally introduce the task we study and how we evaluate it.

[1] If the interruption happens in R_i, the user is currently speaking the $(i+1)$-th speech chunk S_{i+1}, as the thinking R_i happens simultaneously when the user says S_{i+1}. Consequently, we also feed the transcription of S_{i+1} into the judge model when determining whether the interruption is valid. Nevertheless, we do not find the results to differ significantly if we only give the judge model up to the speech chunk S_i.

3.2.1 TASK DESCRIPTION

To show that SLMs can make tool calls when listening to the user request, we adopt ComplexFuncBench (Zhong et al., 2025), an evaluation dataset that assesses LLMs' ability to make multi-step API calls. An instance in ComplexFuncBench includes a user query in text specifying some requirements for a travel plan and a list of tool descriptions that are required to complete the task. Some example tools include APIs for searching hotels or flights.

An evaluated model needs to make relevant API calls and provide the resulting information to the user. A single user query typically requires multiple API calls, and these calls may have dependencies in which the output of one call becomes an argument to a subsequent call, so the call order matters and some calls may not be run in parallel. For each instance, the dataset provides the ground truth API calls and their corresponding API responses, which can be used to evaluate whether the model's API call is correct. To adapt ComplexFuncBench to our spoken-dialogue setting, we use GPT-4o-mini-TTS to synthesize the user's spoken query from the text instructions from ComplexFuncBench.

3.2.2 TRAINING DATA FOR TOOL CALL

To train SHANKS to perform Tool calls when listening, we need to teach the model to make API calls in their thinking process R_i based on user input speech chunk S_i. We split half of the instances in ComplexFuncBench to construct the training data and the other half as the testing data.[2] The speech chunks S_i in the training data can simply be obtained from segmenting the audio of the user query speech.

The next step is to construct the thinking chunks R_i. In this task, **the thinking chunk R_i is the API calls and call responses**. For each user query, ComplexFuncBench already provides the ground truth API calls to complete the task, and we only need to determine which API calls, among the ground truth API calls, can be made after a speech chunk S_i. To determine which API calls can be made in R_i, recall that a thinking chunk R_i is based on the user speech up to $i \times t_{\text{chunk}}$ seconds, so as long as the speech up to $i \times t_{\text{chunk}}$ provides the information to make an ground truth API call, that API call can be included in R_i. We follow the above idea and prompt GPT-o1 (OpenAI, 2024c) with the transcription of the user speech, the time alignment of each word in the user's speech, and the ground truth APIs, and ask GPT-o1 to determine the earliest time an API call can be made. The prompt is shown in Table 8 in the Appendix. Based on the above process, we can determine which ground truth APIs should be included in which thinking chunk R_i. A thinking chunk R_i can have more than one API call and response. If no API calls can be made in R_i, we put a template message that says there are no additional tool calls that can be made currently.

The final response O is also generated with GPT-4o by prompting it to generate a final response based on the user query, all the ground truth API calls, and the corresponding responses. During training, the descriptions of the API calls necessary to complete the user's request will be included in the system prompt to let the model know what APIs can be used.

An illustration of a training instance is shown in Figure 2(c). During training, the loss for the API call response in R_i is masked. Training on this dataset will teach the model to make API calls based on incomplete user queries as long as the information for an API call is sufficient.

3.2.3 INFERENCE AND EVALUATION

During inference, for a testing instance, the model is given the user's speech in a streaming manner; the descriptions of the APIs that can be used in this testing instance. When the model makes an API call, we use GPT-4o as a judge to determine if the API call matches one of the ground truth API calls, and return the response of the ground truth API call if a match is found. If the API call does not match the API call in the ground truth, we return a generic error message. Using GPT-4o to match the API call made by the model against the ground truth follows one of the evaluation protocols in the original ComplexFuncBench (Zhong et al., 2025).

[2] ComplexFuncBench is originally designed as an evaluation dataset. Here, we train the model directly on this dataset since the models we use were not trained to perform tool call. Since our training data has the **exactly same** distribution as the testing data, our results should not be compared with other models that are not trained on this dataset.

The testing set includes 500 instances, and each testing instance requires 5.1 API calls to complete on average. The average duration of the audio of the user's speech is 18.71 seconds. We evaluate the performance using five metrics:

1. **Call accuracy**: The number of ground truth API calls that are successfully made by the model, divided by the total number of ground truth API calls. We also calculate the *early call accuracy*, defined as the ground truth API calls that are successfully made *when the user is still speaking*, divided by the total number of ground truth API calls. Similarly, we calculate the *late call accuracy*, where the dividend is the ground truth API calls that are successfully made *after the user finishes speaking.* This helps us understand how well the model uses the time when the user is still speaking; this can be used as a proxy to measure latency.
2. **Success rate**: The percentage of user queries that are successfully completed. If all the ground truth API calls for a user query are successfully made, the task is considered successful.
3. **Correctness**: We evaluate if the final response O is accurate and aligns with the API call responses. Following Zhong et al. (2025), we use GPT-4o to give a score in $\{0, 1, 2\}$, indicating if the transcription of the response O is completely incorrect, partially correct, or completely correct the user's request.
4. **Completeness**: We evaluate if the final response O fully satisfies the user's request. Following Zhong et al. (2025), we use GPT-4o to give a score in $\{0, 1, 2\}$, indicating if the transcription of the response O does not satisfy, partially satisfies, or fully satisfies the user's request.
5. **Latency**: We evaluate the response latency by the number of tokens the model generates before it emits the final response O and after the user has finished their turn. We choose to report the token count instead of the exact time since the precise time to emit these tokens depends on hardware and implementation.

4 Compared Methods

In this section, we introduce the models that we will compare in our experiments, including two variants of the SHANKS models. Additionally, for each task in Section 3.1 and 3.2, we include a scenario-specific baseline model. The training details and hyperparameters are included in Appendix B.

4.1 SHANKS-E2E

We fine-tune an end-to-end (E2E) SLM to make it able to think while listening. We will use Qwen-2.5-Omni (Qwen-omni for short) (Xu et al., 2025a), one of the best open-sourced end-to-end SLMs, in our experiment. Qwen-omni is a *thinker-talker SLM*. The thinker takes speech representation extracted by a speech encoder (Chu et al., 2024) as the input and generates **text tokens**. The talker functions like a text-to-speech (TTS) model, taking the hidden representation from the thinker as the input and generating the output speech.

Originally, Qwen-omni is not capable of performing *unspoken* thinking – every token (and its corresponding hidden representation) generated by the thinker will be sent to the talker model and synthesized into speech. After fine-tuning the thinker model on the training dataset mentioned before, the model will learn to enclose the unspoken thinking process within `<think>` and `</think>`. Since we only want the Qwen-omni to speak out the response tokens, we only pass the response tokens and their hidden states to the talker.

As stated in Section 2.1, SHANKS uses the time when the user is speaking the next speech chunk S_{i+1} to generate the thinking chunk R_i, so the number of thinking tokens in R_i cannot exceed $t_{\text{chunk}} \times n_{\text{tps}}$, where n_{tps} is the number of tokens the model can generate per second.[3]

[3]The tokens in the API call responses are not included in the $t_{\text{chunk}} \times n_{\text{tps}}$ limit since these are not the tokens generated by the SLM itself. For simplicity, we do not consider the API call latency, as our environment already prepares all the ground truth API responses.

4.2 SHANKS-CASCADE

We set up a cascade version of SHANKS. Precisely, we cascade an ASR (Whisper-large-v3 (Radford et al., 2023)) with a stronger text-only LLM, Qwen-2.5-7B-Instruct (Qwen et al., 2025), to make the LLM generate thinking chunks while reading the partial transcription. Qwen-2.5-7B-Instruct and Qwen-omni are fine-tuned from the same base model, while Qwen-2.5-7B-Instruct are fine-tuned on a much larger reasoning dataset. This baseline allows us to know what the performance of SHANKS can be if we use a model with better reasoning ability as the backbone. The training data of SHANKS-E2E and Cascade are almost the same, only differing in the input modality.

4.3 SCENARIO-SPECIFIC BASELINES

4.3.1 BASELINE FOR INTERRUPTION

We fine-tune a baseline model using Qwen-omni, which we name "**No-thinking**". We fine-tune it to predict whether it should interrupt the user without any thinking. The model is trained to predict special tokens, `[NO_INTERRUPT]` or `[INTERRUPT]`, to indicate whether the model should interrupt the user, given chunked user input speech. This can be thought of as SHANKS while the thinking chunks only contain a `[NO_INTERRUPT]` or `[INTERRUPT]` special token.

We do not compare with other models since there are no other models that can interrupt the user. While some full-duplex SLMs should be able to interrupt the user by design (Défossez et al., 2024), we find that these models cannot interrupt the user at all. We also find that closed-source models like GPT-4o cannot interrupt the user when the user is still talking.

4.3.2 BASELINE FOR TOOL CALL

For this baseline, we fine-tune a model using Qwen-omni that takes the *full* user speech and then makes all the API calls. We call this model "**Call-after-listen**". This serves as a baseline that waits until receiving the full user input and then begins to make API calls; this is how existing tool-augmented models operate. During inference, this model takes the complete user query and then iteratively makes API calls and receives the responses until the model thinks all necessary API calls are made, and then generates the final response.

5 EXPERIMENT RESULTS

5.1 RESULTS FOR SCENARIO 1: INTERRUPTING USER TURN

The results for interrupting the user turn are presented in Table 1. We have the following observations.

SHANKS is more likely to interrupt on the wrong subset. Comparing the interruption ratio of SHANKS on the correct and wrong subsets, the interruption ratio is 54.2% higher on the wrong subset. This shows that SHANKS is indeed capable of capturing the errors in the user's speech and interrupt appropriately. Based on the valid interruption ratio for the wrong subset, about 2 out of 3 interruptions made by SHANKS are valid. Interesting, on the correct subset, the valid interruption ratio is non-zero. By looking into the instances in the correct subset, we find that even if their final answers are correct, sometimes their intermediate reasoning may be odd or ambiguous, and the model will interrupt and ask for clarification. Prior works also reported that even if the final answer of the model is correct, the CoT reasoning may be wrong (Golovneva et al., 2023). In this case, the LLM judge treats this kind of interruption as valid.

SHANKS' interruption latency shows that the model mostly interrupts after the error occurs. On the wrong subset, the interruption latency of SHANKS-E2E is 5.08 seconds on average. In Figure 5 in the Appendix, we further plot the distribution of the interruption latency. We find that the interruption latency on the wrong dataset is left-skewed, where more samples fall on the right proportion of the distribution and have a positive interruption latency. This indicates that most interruption happens later than the first error.

A qualitative example in Figure 3 shows that SHANKS-E2E can interrupt the user when there is a mistake. To allow the readers to have a better idea of how SHANKS interrupts the user, we show

Table 1: Results for interrupting the user. We report the interruption ratio and valid interruption ratio in percentage, and the interruption latency in seconds. $t_{\text{chunk}} = 4.0$ in the top three rows.

Methods	**Correct Subset** (1280)		**Wrong Subset** (1140)		
	Interrupt ratio (%) (↓)	Valid interrupt ratio (%) (↑)	Interrupt ratio (%) (↑)	Valid interrupt ratio (%) (↑)	Interruption latency (s)
No-thinking	1.4%	16.7%	13.8%	26.8%	6.46
SHANKS-E2E	30.6%	25.7%	84.8%	63.9%	5.08
SHANKS-Cascade	24.9%	40.3%	86.1%	78.3%	6.90
Ablations for SHANKS-*E2E*					
$t_{\text{chunk}} = 3$	41.1%	21.4%	88.7%	60.3%	1.56
$t_{\text{chunk}} = 5$	26.9%	36.9%	83.1%	66.2%	8.19

an example in Figure 3. When the user unfolds the question, SHANKS already starts thinking about the math question. For example, when $t \in [4t_{\text{chunk}}, 5t_{\text{chunk}}]$, the model's thinking already calculates several intermediate variables, including the number of total petunias and sweet potato vines. When the user finishes stating the question after $t = 6t_{\text{chunk}}$, the model already completes the calculation and has the correct answer in its mind. The model also correctly identifies the user's error in falsely calculating that there are 25 petunias and interrupts the user during $t \in [12t_{\text{chunk}}, 13t_{\text{chunk}}]$.

Interruption without thinking leads to much poorer performance. The performance of the no-thinking baseline is much worse than SHANKS, which performs reasoning before interrupting. The no-thinking baseline has a much lower interruption ratio on the wrong subset, and the valid interrupt ratio is also much lower than SHANKS. This shows that thinking before interruption is important, justifying the design of SHANKS.

Cascade version of SHANKS with stronger LLM leads to the best performance. When using Qwen-2.5-7B-Instruct as the backbone model for SHANKS, the performance can be even better. The interruption ratio on the correct subset is lower, and the valid interruption ratio on the wrong subset also grows higher. This shows that the interruption ability of SHANKS is mostly related to the reasoning ability of the backbone model, and using a stronger reasoning LLM can improve the performance.

Varying t_{chunk} at inference time does not significantly affect the performance. When constructing the training data, we fix $t_{\text{chunk}} = 4$ seconds. Here, we ask whether we can vary t_{chunk} at inference time. Since the thinking of SHANKS always lags behind the latest user speech by t_{chunk} seconds, changing t_{chunk} can affect how soon the SLM can hear the latest user speech and affect the response latency. As an ablation, we change t_{chunk} to 3 and 5 during inference without retraining SHANKS-E2E. The results are shown in the bottom two rows in Table 1. On the wrong subset, we do not find the interrupt ratio and valid interrupt ratio to change significantly compared with $t_{\text{chunk}} = 4$. Interestingly, we find that the interrupt latency on the wrong subset for $t_{\text{chunk}} = 3$ is the smallest, while the $t_{\text{chunk}} = 5$ has the largest interrupt latency.

5.2 RESULTS FOR SCENARIO 2: MAKING TOOL CALLS WHEN LISTENING

Next, we move on to the second scenario: making tool calls when listening. The experiment results are presented in Table 2. We summarize the main findings as follows:

SHANKS successfully makes at least 56.9% of API calls when the user is still speaking. Among the successful API calls made by the two SHANKS models, about 80% to 90% of the API calls are made during the user speech. In the example in Figure 4, SHANKS-E2E makes four out of six API calls when the user is still speaking. Compared to the call-after-listen baseline, which makes all the API calls after the user turn finishes, SHANKS can significantly reduce the response latency by using the time the user is speaking to make tool calls.

Call-after-listen has a higher success rate and response quality. While SHANKS elegantly uses the user speaking time to make API calls, the success rate and response quality (correctness and

Figure 3: An example from the interruption scenario in Section 3.2. The chunks in red are the transcriptions of a user describing a math problem and attempting to solve it step-by-step. The thinking chunks (in green) and interruption response (in orange) are generated by SHANKS-E2E. For each time slot from nt_{chunk} to $(n+1)t_{\text{chunk}}$, the chunks in green (SLM thinking chunks) and orange (output response) happen *sequentially*, while the user speech chunk (in red) happens concurrent to other blocks in the same time slot.

completeness) lag behind call-after listen. We find that this is because during inference, if the API call fails during R_i, SHANKS seldom retries the failed API call in future R_j, where $j > i$. On the other hand, the call-after-listen baseline is more likely to retry failed API calls.

Combining SHANKS and call-after-listen yields the best performance. To solve the previously observed issue, a simple method is to use SHANKS when the user is still speaking and back off to call-after-listening when the user's speech ends. Precisely, when the user is still speaking, we use

Table 2: Results for API calls. "Early" and "Late" are computed over the total set of ground-truth API calls; they need not sum to 100% because some calls may be incorrect.

Method	Call accuracy (%)			Success rate (%)	Correctness (0-2)	Completeness (0-2)	Latency (token)
	Early	Late	Total				
Call-after-listen	0.0	86.5	86.5	63.2	1.17	1.37	313
SHANKS-E2E	56.9	14.4	71.3	35.2	0.79	1.00	54
SHANKS-CASCADE	63.9	5.4	69.3	34.9	0.73	1.00	153
SHANKS+Call-after-listen	57.3	32.8	90.0	62.3	1.31	1.43	117

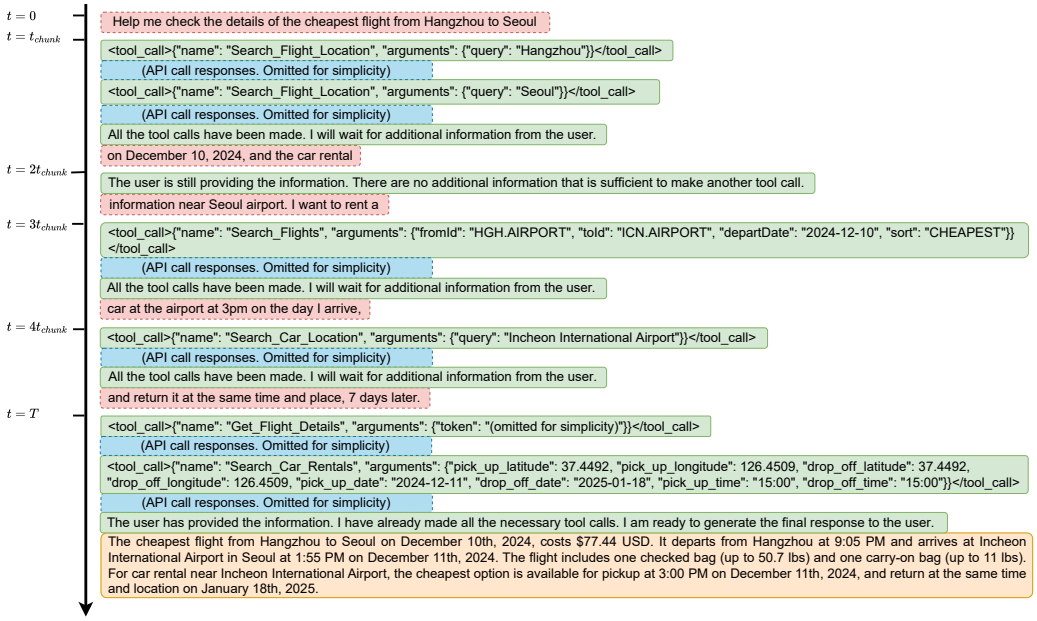

Figure 4: An example user query from ComplexFuncBench (in red), including the unspoken thinking process (in green) and the spoken final response (in orange) from SHANKS-E2E. For each time slot from nt_{chunk} to $(n+1)t_{\text{chunk}}$, the chunks in green (SLM thinking chunks), blue (API call responses), and orange (output response) happen *sequentially*, while the user speech chunk (in red) happens concurrent to other blocks in the same time slot. The $t = T$ means the time when the user's speech terminates.

SHANKS to call APIs while listening, and only keep the successful API calls and their responses. When the user finishes their speech, we switch to the call-after-listening mode, where the input to the SLM is the complete user speech. We also feed the success API calls and responses previously made by SHANKS to the model, as if the call-after-listen model had made those API calls by itself. Based on the full input speech and previous success API calls and responses, the model continues to make the remaining API calls. Since some API calls have already been made by SHANKS when the user speaks, this combined method enjoys the thinking-while-listening advantage of SHANKS.

In the last row in Table 2, we show the result of combining SHANKS-E2E with call-after-listening. This combined method has a high number of early call accuracy while also having a higher task success rate and response quality. Among the API calls that are successful, over 60% of API calls are made during the user speech, while only 40% of the API calls are made after the user finishes. This is in stark contrast to call-after-listen, where 100% of the successful API calls are made after the user finishes, resulting in a much higher latency. As a result, the combined method only needs to generate on average 117 tokens after the user has finished, while the call-after-listen baseline needs to generate on average 313 tokens after the user's turn ends. So thinking while listening reduce the response by 62.3%.

6 Related Works

Thinking before responding has been widely explored in text-only LLMs (OpenAI, 2024c; Guo et al., 2025). Recently, this "*think then respond*" paradigm has been applied to audio-aware language models, which takes speech (or audio) as the input and output texts (Xie et al., 2025). Note that these audio-aware language models, which only output text, are different from the speech-in-speech-out SLMs focused on in our paper. While thinking before responding can improve the response quality and yield significant performance on many challenging benchmarks (Lightman et al., 2024; Sakshi et al., 2025), thinking before responding creates great response latency. As a result, it is impractical to directly apply thinking-before-responding to SLMs, speech-in-speech-out models, which require real-time and low-latency interaction (Li et al., 2025; Xu et al., 2025a;b). Developing reasoning methods that preserve real-time interaction in SLMs remains an open problem.

Concurrent to us, Chiang et al. (2025) introduce thinking to SLMs by a *thinking-while-speaking* method called STITCH. STITCH uses the fact that a chunk of audio in the speech response takes less time to generate than it does to play to the user, and the model can use the remaining time to generate thinking tokens when the SLM is still speaking. While both SHANKS and STITCH explore unspoken thinking processes for SLMs, the main distinction is *when* the thinking happens. In SHANKS, the thinking process happens when the user is still speaking, while STITCH thinks when the SLM is speaking. In fact, the two methods can be combined: an SLM can think when listening and speaking. We believe this will be the future of SLM, and we leave this as a promising future direction.

Another concurrent work released on arXiv less than one week ago (10/02/2025), Stream RAG (Arora et al., 2025), also studies calling tools (web search and knowledge graph APIs) during the user's speech. This is similar to our second scenario introduced in Section 3.2. However, Stream RAG focuses on when to issue retrieval/tool queries while listening and does not introduce an explicit silent chain-of-thought ('thinking') process like we do. In contrast, our paper studies a broader *thinking-while-listening* paradigm, with tool-calling as one application, and show benefits such as improved user-interruption decisions.

7 Conclusion, Limitations, and Future Work

In this paper, we introduce SHANKS, a framework that enables SLMs to think while listening. SHANKS achieves thinking-while-listening by chunking the user input speech and progressively reasoning over the available user inputs. When the user is speaking, SHANKS is generating thinking chunks for all previous input speech, achieving thinking while listening. We demonstrate the potential of SHANKS on two scenarios: First, SHANKS can listen to the user solving a math problem step-by-step, and interrupt the user when the user is making a mistake. Second, we focus on a tool-augmented task-oriented dialogue setting and show that SHANKS can listen to the user speech and evoke necessary API calls when the user is still speaking. On ComplexFunxBench, SHANKS successfully calls more than half of the APIs that are required to complete the user's request when the user is still speaking. This reduces the response latency, as the model only needs to call the remaining half APIs after the user has finished.

While SHANKS shows great potential in improving the user-SLM interaction, we see the following limitations of the method. First, SHANKS requires the user's speech to have certain structures: the user's speech needs to be long enough to allow the model to perform meaningful reasoning when listening, and the information in the user's speech needs to be able to be processed in a sequential order. This kind of speech can naturally occur, as shown in the two scenarios we studied.

Next, SHANKS uses a fixed chunk size to segment the user input speech. The chunking nature of SHANKS means that SHANKS always lags behind the user's speech by t_{chunk} seconds, incurring latency in the thinking process. We encourage future work to reduce the latency between thinking and listening by using more sophisticated chunking methods.

Last, the goal of the user might be unclear when the user's speech is not completed, and the thinking tokens generated during listening may be redundant and not always be useful to address the goal of the user. While thinking during listening does not incur additional latency after the user's speech ends, SHANKS still significantly increases the compute cost during inference. Although SHANKS has some limitations, we believe that our effort in proposing a novel modeling method, thinking while

listening, together with the reasonable scenarios and convincing results, already contributes greatly to the research community by shedding light on a potentially fruitful research direction.

ACKNOWLEDGMENTS

The authors would like to thank Ke-Han Lu, Chen-An Lee, Yi-Cheng Lin, and Wei-Chih Chen for their valuable feedback on the draft of this paper.

REFERENCES

Siddhant Arora, Haidar Khan, Kai Sun, Xin Luna Dong, Sajal Choudhary, Seungwhan Moon, Xinyuan Zhang, Adithya Sagar, Surya Teja Appini, Kaushik Patnaik, et al. Stream rag: Instant and accurate spoken dialogue systems with streaming tool usage. *arXiv preprint arXiv:2510.02044*, 2025.

Sara Bögels, Lilla Magyari, and Stephen C Levinson. Neural signatures of response planning occur midway through an incoming question in conversation. *Scientific reports*, 5(1):12881, 2015.

Sara Bögels, Marisa Casillas, and Stephen C Levinson. Planning versus comprehension in turn-taking: Fast responders show reduced anticipatory processing of the question. *Neuropsychologia*, 109: 295–310, 2018.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

Cheng-Han Chiang and Hung-yi Lee. Can large language models be an alternative to human evaluations? In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15607–15631, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.870. URL https://aclanthology.org/2023.acl-long.870/.

Cheng-Han Chiang, Xiaofei Wang, Linjie Li, Chung-Ching Lin, Kevin Lin, Shujie Liu, Zhendong Wang, Zhengyuan Yang, Hung-yi Lee, and Lijuan Wang. Stitch: Simultaneous thinking and talking with chunked reasoning for spoken language models. *arXiv preprint arXiv:2507.15375*, 2025.

Yunfei Chu, Jin Xu, Qian Yang, Haojie Wei, Xipin Wei, Zhifang Guo, Yichong Leng, Yuanjun Lv, Jinzheng He, Junyang Lin, et al. Qwen2-audio technical report. *arXiv preprint arXiv:2407.10759*, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Ruth E Corps, Abigail Crossley, Chiara Gambi, and Martin J Pickering. Early preparation during turn-taking: Listeners use content predictions to determine what to say but not when to say it. *Cognition*, 175:77–95, 2018.

Alexandre Défossez, Laurent Mazaré, Manu Orsini, Amélie Royer, Patrick Pérez, Hervé Jégou, Edouard Grave, and Neil Zeghidour. Moshi: a speech-text foundation model for real-time dialogue. *arXiv preprint arXiv:2410.00037*, 2024.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024. URL https://arxiv.org/abs/2312.10997.

Olga Golovneva, Moya Peng Chen, Spencer Poff, Martin Corredor, Luke Zettlemoyer, Maryam Fazel-Zarandi, and Asli Celikyilmaz. Roscoe: A suite of metrics for scoring step-by-step reasoning. In *The Eleventh International Conference on Learning Representations*, 2023.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, Joe Davison, Mario Šaško, Gunjan Chhablani, Bhavitvya Malik, Simon Brandeis, Teven Le Scao, Victor Sanh, Canwen Xu, Nicolas Patry, Angelina McMillan-Major, Philipp Schmid, Sylvain Gugger, Clément Delangue, Théo Matussière, Lysandre Debut, Stas Bekman, Pierric Cistac, Thibault Goehringer, Victor Mustar, François Lagunas, Alexander Rush, and Thomas Wolf. Datasets: A community library for natural language processing. In Heike Adel and Shuming Shi (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 175–184, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-demo.21. URL https://aclanthology.org/2021.emnlp-demo.21/.

Tianpeng Li, Jun Liu, Tao Zhang, Yuanbo Fang, Da Pan, Mingrui Wang, Zheng Liang, Zehuan Li, Mingan Lin, Guosheng Dong, et al. Baichuan-audio: A unified framework for end-to-end speech interaction. *arXiv preprint arXiv:2502.17239*, 2025.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=v8L0pN6EOi.

Guan-Ting Lin, Jiachen Lian, Tingle Li, Qirui Wang, Gopala Anumanchipalli, Alexander H. Liu, and Hung yi Lee. Full-duplex-bench: A benchmark to evaluate full-duplex spoken dialogue models on turn-taking capabilities, 2025. URL https://arxiv.org/abs/2503.04721.

Ilya Loshchilov and Frank Hutter. SGDR: Stochastic gradient descent with warm restarts. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=Skq89Scxx.

Hongyin Luo, Tianhua Zhang, Yung-Sung Chuang, Yuan Gong, Yoon Kim, Xixin Wu, Helen Meng, and James Glass. Search augmented instruction learning. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3717–3729, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.242. URL https://aclanthology.org/2023.findings-emnlp.242/.

OpenAI. Introducing next-generation audio models in the api, 2024a. URL https://openai.com/index/introducing-our-next-generation-audio-models/. Accessed on May 12, 2025.

OpenAI. Hello gpt-4o, 2024b. URL https://openai.com/index/hello-gpt-4o/. Accessed on May 12, 2025.

OpenAI. Learning to reason with llms, 2024c. URL https://openai.com/index/learning-to-reason-with-llms/. Accessed on July 15, 2025.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. Tool learning with foundation models, 2024a. URL https://arxiv.org/abs/2304.08354.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=dHng2O0Jjr.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022. URL https://arxiv.org/abs/2212.04356.

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International conference on machine learning*, pp. 28492–28518. PMLR, 2023.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=Ti67584b98.

S Sakshi, Utkarsh Tyagi, Sonal Kumar, Ashish Seth, Ramaneswaran Selvakumar, Oriol Nieto, Ramani Duraiswami, Sreyan Ghosh, and Dinesh Manocha. MMAU: A massive multi-task audio understanding and reasoning benchmark. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=TeVAZXr3yv.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Yacmpz84TH.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2609–2634, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.147. URL https://aclanthology.org/2023.acl-long.147/.

Mingqiu Wang, Izhak Shafran, Hagen Soltau, Wei Han, Yuan Cao, Dian Yu, and Laurent El Shafey. Retrieval augmented end-to-end spoken dialog models. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 12056–12060. IEEE, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Zhifei Xie, Mingbao Lin, Zihang Liu, Pengcheng Wu, Shuicheng Yan, and Chunyan Miao. Audio-reasoner: Improving reasoning capability in large audio language models. *arXiv preprint arXiv:2503.02318*, 2025.

Jin Xu, Zhifang Guo, Jinzheng He, Hangrui Hu, Ting He, Shuai Bai, Keqin Chen, Jialin Wang, Yang Fan, Kai Dang, et al. Qwen2. 5-omni technical report. *arXiv preprint arXiv:2503.20215*, 2025a.

Jin Xu, Zhifang Guo, Hangrui Hu, Yunfei Chu, Xiong Wang, Jinzheng He, Yuxuan Wang, Xian Shi, Ting He, Xinfa Zhu, et al. Qwen3-omni technical report. *arXiv preprint arXiv:2509.17765*, 2025b.

Weilin Zhao, Yuxiang Huang, Xu Han, Wang Xu, Chaojun Xiao, Xinrong Zhang, Yewei Fang, Kaihuo Zhang, Zhiyuan Liu, and Maosong Sun. Ouroboros: Generating longer drafts phrase by phrase for faster speculative decoding. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 13378–13393, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.742. URL https://aclanthology.org/2024.emnlp-main.742/.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In Yixin Cao, Yang Feng, and Deyi Xiong (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 400–410, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-demos.38. URL https://aclanthology.org/2024.acl-demos.38/.

Lucen Zhong, Zhengxiao Du, Xiaohan Zhang, Haiyi Hu, and Jie Tang. Complexfuncbench: exploring multi-step and constrained function calling under long-context scenario. *arXiv preprint arXiv:2501.10132*, 2025.

A Encoding the User Speech

In the main content of the paper, we say that we chunk the user input audio into fixed-size chunks of t_{chunk} seconds. In fact, what we do is chunking at the level of feature representation instead of the level of the audio waveform. Precisely, when encoding the $i \geq 2$ speech chunks S_i, we feed the full speech through the audio encoder, and only take the speech representation for the corresponding speech chunk. If we directly chunk the audio waveform and encode each audio chunk independently, the representation of later audio chunks will not be able to depend on the earlier audio chunks, which can potentially lead to performance degradation.

B Details in Training

We fine-tune the models using the Llamafactory (Zheng et al., 2024) toolkit. When generating the training data using GPT-4o, we do not feed the audio of the user's speech into GPT-4o. Instead, we feed the transcription of the speech chunks. This is because using the speech chunk will increase the cost and time to call the API. To obtain the transcription of each chunk, we use Whisper-large-v3 (Radford et al., 2023) to obtain the transcription and timestamp for each word in the user speech, and then segment the transcriptions into chunks based on the timestamp. While the timestamp obtained from Whisper may not be very precise, this is already sufficient for preparing the training data.

B.1 Fine-tuning for Interruption

To prepare the training data, we randomly sample 5K samples from Tulu-3-SFT-Math-Grade (Lambert et al., 2024), which can be loaded from Huggingface datasets (Lhoest et al., 2021): `https://huggingface.co/datasets/allenai/tulu-3-sft-personas-math-grade`. We follow the procedure detailed in Section 3.1.2 to construct the training data. We additionally filter out audios that are longer than 80 seconds, so the final training dataset is slightly less than 5K.

We fine-tune the thinker on the training data for two epochs on 8 A100 GPUs. The effective batch size is 64. We set the learning rate to $1.0e-4$ with cosine learning rate scheduling and a 0.1 warm-up ratio (Loshchilov & Hutter, 2017). The same training hyperparameters are used across all three models, including the SHANKS-E2E, SHANKS-Cascade, and no-thinking model.

The training data is mostly generated by GPT-4o. We include the prompts to generate the reasoning chunks in Table 3 and 4, the prompt to generate the interruption in Table 5, and the prompt to generate the response without interruption in Table 6.

B.2 Fine-tuning for Tool Calls

We use the procedure detailed in Section 3.2.2 to construct the training data. The training data consists of 500 samples. The prompt used to determine when an API call can be made is shown in Table 8. The prompt used to generate the final response is shown in Table 9.

For the think-after-listen and SHANKS-E2E model, we fine-tune them using LoRA (Hu et al., 2022), as the sequence length for this dataset is very large and full fine-tuning will result in out-of-memory. We also fine-tune the LM head and the token embedding of the talker model; otherwise, the model will not be able to recognize and generate special tokens. For the SHANKS-Cascade model, we fine-tune all the parameters. As the training dataset is smaller, we fine-tune the model for 10 epochs, while other training hyperparameters follow those in Appendix B.1.

C Details in Evaluation

C.1 Evaluation Details for Interruption

To determine the time of interruption $t_{\text{interrupt}}$ when evaluating the interrupt latency, we apply the following procedure. We use Whisper-large (Radford et al., 2022) to obtain the timestamp of each word in the user speech from the testing set, and we use GPT-4o to determine when the first error in the user speech occurs by giving GPT-4o the question, the ground truth answer, the transcription of the user speech, and the word-timestamp alignment. The prompt used to determine the first error time t_{error} is shown in Table 7.

To evaluate the valid interrupt ratio, we use GPT-4o as the judge. The prompt used to determine whether an interruption is valid is shown in Table 10.

C.2 Evaluation Details for Tool Call

When evaluating the SHANKS models, when the user is still speaking, each thinking chunk can only include at most 320 tokens generated by the SLM itself. In some cases, the API call augments may include very long tokens, which can easily exceed the token limit for the thinking chunk, and the

API call will be incomplete and unsuccessful. While we do not specifically handle this kind of case, there is a simple workaround that can resolve the above issue: The long arguments for the API call generated by the model must be the returned value of previous tool calls, so we can include previous API call responses in a reservoir of the draft for speculative decoding (Leviathan et al., 2023; Zhao et al., 2024), and use speculative decoding to speed up the inference speed.

ComplexFuncBench is originally designed to evaluate a model's ability to parse long-context information, and the API call response can be very long. However, since Qwen-omni only has a context length of 32,768, once the token exceeds this limit, we directly terminate the inference for a tested instance. As a result, some of the testing samples we evaluate may fail because the number of tokens exceeds the max sequence length of the model.

Figure 5: The interruption latency for SHANKS. The bars in red are the results on the wrong subset, while the bars in green are the results on the correct subset. One can observe that the red bars are mostly positive, meaning that the model tends to interrupt after the first error occurs.

```
# Generate Internal Thinking While Listening

## Task Introduction
Humans are capable of thinking while listening to others speak.  Based on the partial information received, we parse important
details, clarify ambiguities, recall relevant facts, and compute intermediate variables.  Your task is to simulate this
process.  You will be the previous chunks of user's speech in text, and you will also see your previous inner thinking when
listening to those chunks.  Your job is to generate the next internal thinking as if you had listened up to the newest chunk.

When generating internal thinking spans, follow these guidelines:
1.  The inner thinking span should be fewer than 400 words.
2.  Your internal thinking should reflect the user's emotion, intent, and what you already know from the user.  If any
relevant information can be recalled or intermediate variables can be calculated based on current information, include them
in your inner thinking.
3.  The inner thinking should read more like full, coherent sentences rather than shorthand notes.  Using short notes will be
very hard to understand and possibly making logical errors.
4.  If the user's query involves a question, you **must generate your own step-by-step answer in the internal thinking before
the user finishes speaking**.
5.  Later internal thinking spans must not repeat information already covered in earlier ones.  However, if later
transcription spans update or contradict earlier information, explicitly point that out and correct it.  You may start with
phrases like \Wait, the user previously..., but now...".
6.  Always think independently in your internal thinking.  When the user is providing there solution, you should have you own
solution and then compare your own solution with the user's solution.  If you identify any error, you should interrupt the
user immediately.  Indicate the interruption by ending your internal thinking with the special token [INTERRUPT].

---

## Samples

### Example 1:
User (partial) input transcription 1
Betty is saving money for a new wallet which

Prior Inner Thinking 1
The user's tone is neutral.  The user describes a situation where someone named Betty is saving money for a new wallet.  The
user hasn't finished yet.  Perhaps they want me to give advice on how to save money.

User (partial) input transcription 2
costs $100.  Betty has only half of the money she needs.

Prior Inner Thinking 2
Now the user gives more information.  We know the wallet Betty wants to buy costs $100, and she has only half of that.  I can
calculate this:  $100 / 2 = $50, so she currently has $50.  The user's intent is still unclear.

User (partial) input transcription 3
Her parents decided to give her $15 for that

Inner thinking to Generate
Well, it seems that Betty's parents will contribute as well.  She will get $15 from them.  Now she has $50 + $15 = $65.  She
still needs $100 - $65 = $35.  It seems the user wants to know how to plan Betty's budget.

### Example 2:
User (partial) input transcription 1
I want to answer the following math question:  Is the function f(x) =

Prior Inner Thinking 1
The user uses a neutral tone and describes a math question they want to solve.  I will try to solve the question when the user
is speaking.  I can verify if the user's solution is correct by comparing my solution with the user's solution.  The question
is about a function in x.  The user has not finished yet.  I will wait until they complete the question.

User (partial) input transcription 2
4x^2 + 4x + 1 always above the x axis?

Prior Inner Thinking 2
The user has finished describing the question.  They want to determine if the function f(x) = 4x^2 + 4x + 1 is always above
the x-axis.  I am ready to answer this question.  Let's compute the discriminant:  D = b^2 - 4ac = 4^2 - 4 * 4 * 1 = 16 -
16 = 0.  The function touches the x-axis at one point.  I have the solution.  Since the user wants to solve the question
by themselves, I will not interrupt the user and listen to the solution.  However, I will interrupt the use if there is any
mistakes.

User (partial) input transcription 3
Here is my solution:  Based on the formula D = b^2 - 2ac,

Inner thinking to Generate
Now I see that the user wants to solve the question themselves.  I should verify whether their solution is correct.  The user
says the formula is D = b^2 - 2ac, but that's incorrect.  The correct formula is D = b^2 - 4ac.  The user has made a mistake,
and I need to interrupt them.  [INTERRUPT]
```

Table 3: Prompt for generating the thinking chunks in the training data for the interruption experiment (part 1).

```
### Example 3:
User (partial) input transcription 1
Bella bought stamps at the post office.  Some of the

Prior Inner Thinking 1
The user uses a neutral tone and describes a math question he wants to solve.  The
question is to The user is introducing a word problem involving someone named Bella
and her purchase of stamps.  It seems like a math problem, likely about counting.
The exact numbers and relationships haven't been shared yet.  I will hold off making
any calculations until I know more about the stamps.

User (partial) input transcription 2
stamps had a snowflake design, some had a truck design,

Prior Inner Thinking 2
We now know there several types of stamps:  snowflake and truck.  There seems to
be more, but the user is still speaking.  There are still no numbers provided, so I
can't compute yet.  I will wait for more information.

User (partial) input transcription 3
and some had a rose design.  Bella bought 11

Prior Inner Thinking 3
Now we know that there are three types of stamps:  snowflake, truck, and rose.
Bella bought 11, but it is unclear which this number corresponds to.  I will wait
for the complete detail from the user to be sure before computing.

User (partial) input transcription 4
snowflake stamps.  She bought 9 more truck stamps

Prior Inner Thinking 4
Now I know that Bella bought 11 snowflake stamps.  I am also told she bought 9 more
truck stamps than snowflake stamps.  I can calculate the number first:  she bought
11 + 9 = 20 truck stamps.  The information we have now is:
- Snowflake:  11
- Truck:  20
The user is still talking, and I am waiting for more information.

User (partial) input transcription 5
than snowflake stamps, and 13 fewer rose stamps than

Prior Inner Thinking 5
Now the user states that Bella bought 13 fewer roses than something, but it is
unclear what is compared here.  I will wait until the user finishes.

User (partial) input transcription 6
truck stamps.  How many stamps did Bella buy in all?

Inner thinking to Generate
Now I know that Bella bought 13 fewer roses than the truck stamps.  There are 20
truck stamps, so I can calculate the number of rose stamp is 20 - 13 = 7.  The user
finishes with a question:  total number of stamps.  I already have all counts:
- Snowflake:  11
- Truck:  20
- Rose:  7
Total = 11 + 20 + 7 = 38 stamps.  I have the answer and I can provide it to the
user.

---

This is the end of the examples.  Now, this is the (partial) user input
transcription, and you need to generate a inner thinking.  You do not need to
explain why the inner thinking you generate is a good one.  Simply generate a good
one without explaining it.

{interleaved_transcription_and_thinking}

Inner thinking to generate (Do not generate anything else other than the inner
thinking)
```

Table 4: Prompt for generating the thinking chunks in the training data for the interruption experiment (part 2).

```
# Task:  Interrupt the user to correct an error

A user is talking to an AI assistant.  You will be given a partial user turn.  There is an error in the user turn and the AI
assistant has identified that error.  The AI assistant needs to interrupt the user.

Your job is to generate the response for the AI assistant that interrupts the user's turn.  You will be given:
(1) A (possibly incomplete) user turn
(2) The inner thinking of the AI assistant.  This inner thinking hasn't been spoken out by the AI assistant and is only
silently kept in the assistant's mind.  We provide you this inner thinking for you to better craft a response.

When correcting and interrupting the user, be precise about what the error is and how to correct it.  You only need to
generate the response without saying anything else.  The conversation between the user and the assistant is in spoken form,
so you need to make your response easy to be spoken while not overly informal and colloquial.

## Example

#### User (partial) input
I want to answer the following math question:  Is the function f(x) = 4x^2 + 4x + 1 always above the x axis?  Here is my
solution:  Based on the formula D = b^2 - 2ac, D = 4^2 - 2 * 4 * 1 = 8 > 0

#### Inner thinking of the assistant
The user uses a neutral tone and describes a math question he wants to solve.  The question is to determine if a 2-degree
function is above the x-axis.  f(x) = 4x^2 + 4x + 1.  Let's use D = b^2 - 4ac = 4^2 - 4 * 4 * 1 = 0.  So the function happens
to intersect with x-axis at one point.  I can answer the user if the user wants me to do so.
But wait, the user themselves want to solve the question, and the user says D = b^2 - 2ac, which is clearly wrong.  The
correct formula should be D = b^2 - 4ac.  I should interupt the user here and tell them the correct formula with a friendly
and reminding tone.

#### Assistant Response
Wait, I think the correct formula should be b^2 - 4ac, not b^2 - 2ac.  The coefficient you mentioned was wrong.

## Now it is your turn

#### User (partial) input
{query}

#### Inner thinking of the assistant
{inner_thinking}

#### Assistant Response
<Write the interrupting response here.  Be precise about the error and the correction; keep it concise and easy to speak.  Do
not include anything else.>
```

Table 5: Prompts for generating an interrupting correction response.

```
# Task:  Generate the spoken response given full user turn and assistant's inner
thinking

A user is chatting with a voice assistant.  Your job is to act as the voice
assistant and generate a valid response that fits in the context.  You will be
given:
(1) The full user turn
(2) The inner thinking of the voice assistant.  Note that the voice assistant may
generate the inner thinking when the user hasn't finished, so it is possible that
some contents in the inner thinking is incorrect.

Guidelines:
1.  Do not generate anything else except the response.
2.  The inner thinking might mention a drafted response.  If the drafted response is
still valid considering the full user turn, follow the draft and start the response.
If the draft is invalid considering the full user input, neglect the draft and craft
a response that is suitable.
3.  This is a spoken dialogue.  Keep the response easy to follow for spoken form.
However, there is no need to deliberately use very colloquial words or phrasing,
making things awkward.

## Input

#### Full user input
{query}

#### Inner thinking of the voice assistant
{inner_thinking}

#### Your Response (Act like the voice assistant)
<Write only the final spoken response here>
```

Table 6: Prompts for generating the response for the interruption application.

```
# Task:  Detect the first reasoning or calculation error with timestamps

You a user's query.  In the user query, the user describes a math problem and
then attempt to solve the problem by themselves.  This user query is in a spoken
form, and I provide you with the transcription.  I will also provide you the force
alignment result of the transcription, which corresponds timestamp of each word in
the spoken response.

Your job is to determine where the problem solving process has the first calcaultion
or reasoning error.  In your response, you should solve the math problem by
yourself, and carefully check the spoken response.  When you see the first error
in the spoken response, use the provided timestamp to determine when the first
error happened.  Conclude you respond with:  "First error:  [time]", where 'time'
is the time where the first error happens.  If the user's problem solving process is
completely correct, please use -1 to indicate that there is no error, i.e., "First
error:  -1"

## Example

### User Query
I want to solve the following math question:  Natalia sold clips to 48 of her
friends in April, and then she sold half as many clips in May.  How many clips did
Natalia sell altogether in April and May?  Here is my solution:  Our goal is to
calculate the total number of clips sold in April and May.  In April, she sold 48.
In May, she sold the half of that, which is 96.  So she sold 48 in April plus 96 in
May, making it 144 in total.

### Word-Timestamp
I - 0.00
want - 0.50
...
96.  - 36.50
...
total.  - 44.00

### Correct Answer
72

### Output
The math problem wants to know how many clips Natalia sold in total.  In April, she
sold 48.  In May, she sold half as many, so she sold 48 / 2 = 24.  In total, she
sold 48 + 24 = 72.  In the problem solving process, the user says that 'half of
that, which is 96.'  This is incorrect.  The correct number for May should be 24.
This is where the first error occurs.  Based on the Word-Timestamp information, the
word 96 is emitted at second.
First error:  36.50

## Now, it is your turn.

### User Query
{question}

### Word-Timestamp
{alignment}

### Correct Answer
{answer}

### Output
<Write the reasoning here and conclude with "First error:  [time]">
```

Table 7: Template for detecting the first error in the interruption task.

```
# Task:  Earliest possible time to call tools during a spoken user query

You are given a user spoken query, which requires some tool usage for answering.
You will be given the tool calls (including their parameters) which are useful for
responding to the user query.  You will also be given the timestamp of each word in
the user's utterance.  Your job is to determine the earliest time that a tool call
can be called when the user is speaking.  That is, when the user is still speaking,
the information that has been spoken by the user may already be sufficient enough
to call some of the tools.  Your job is to determine the **earliest time** during
the utterance that a tool call can be called.  A tool can be called if only if it is
clear what tool should be call and what the paramaters are for the tool call.

### User Spoken Query
{question}

### Time Stamp of Each word
{alignment}

### Tools that needs to be called
{tools}

### Total number of tool calls
{count}

### Output Format
Your response should be a python dictionary.  The key of this dictionary is an
integer index of the tool call shown above, and the value is the earliest time the
tool can be called.  Your response should only include a python dictionary.  The
first character in your response should be the left bracket while the last character
in your response should be the right bracket.  Your response should be able to be
directly converted into a python dictionary using eval().  If there are N tools that
need to be called, your output dictionary should have N items.  I also provide you
the number of tool calls, so you should verify if your output dictionary matches the
number of tool calls.

### Your response:
<Return only a python dictionary, e.g., {0:  12.5, 1:  18.0}>
```

Table 8: Template for checking the earliest callable time for an API.

```
# Task:  Generate the final user-facing response from tool call results

You will be given a user query.  The user query can only be responded based on
the results of some external tool call.  I will show you the tool calls and call
responses.  Your task is to generate a final response to the user based on the tool
call results.  The final response to the user should satisfy the user's original
query and omit unnecessary information.  Some intermediate processes in the tool
call may simply be some process to resolve the variables, and they are not necessary
to be included in the final response to the user.

### User Query
{transcription}

### Previous API Calls
{previous_tool_calls}

### Response to the User Query (Only provide the response.  Do not include anything
else.)
<Write only the final user-facing response here, distilled from the tool results and
satisfying the query.  Exclude setup steps and variable-resolution details.>
```

Table 9: Prompts for generating a final response O in the API call application.

```
# Task:  Judge if the assistant's interruption is reasonable

A user is speaking to a voice assistant.  When the user is speaking, the assistant
tries to interrupt the user.  Your job is to judge if the assistant is interrupting
the user in a reasonable way.  A reasonable interruption is when the user says
something wrong and ambiguous, and the assistant is trying to help correct or
clarify the user's statement.

Here is the user's speech before the assistant interrupted:
{user_speech_before_interrupt}

Here is the assistant's speech that attempts to interrupt the user:
{assistant_speech_after_interrupt}

Please judge if the assistant is interrupting the user in a reasonable way.  If the
assistant is interrupting the user in a reasonable way, return "yes".  Otherwise,
return "no".  Please provide some explanation for your judgment and conclude with
"Final verdict:  Yes/No".  A valid interruption is when the user is indeed making a
mistake and the assistant is trying to help correct or clarify the user's statement.

### Output
<Write your explanation here.  Conclude with "Final verdict:  Yes" or "Final
verdict:  No">
```

Table 10: The prompt used for judging whether an interruption is reasonable.