

Asynchronous Tool Usage for Real-Time Agents

Antonio A. Ginart* Naveen Kodali* Jason Lee*
Caiming Xiong Silvio Savarese John Emmons*

Salesforce AI Research

ABSTRACT

While frontier large language models (LLMs) are capable tool-using agents, current AI systems still operate in a strict turn-based fashion, oblivious to passage of time. This synchronous design forces user queries and tool-use to occur sequentially, preventing the systems from multitasking and reducing interactivity. To address this limitation, we introduce *asynchronous AI agents capable of parallel processing and real-time tool-use*. Our key contribution is an event-driven finite-state machine architecture for agent execution and prompting, integrated with automatic speech recognition and text-to-speech. Additionally, we present the first dataset tailored for instruction-tuning LLMs for asynchronous tool-use. Drawing inspiration from the concepts originally developed for real-time operating systems, this work presents both a conceptual framework and practical tools for creating AI agents capable of fluid, multitasking interactions.

1 INTRODUCTION

With the advancement of large-scale foundation models [Tea22] into the terabyte realm, AI models have become sufficiently capable to function as tool-using agents [Tea24]. A crucial component in training these base models to user instructions is supervised instruction tuning [Tea24, OWJ⁺22, PLH⁺23]. In addition, the integration of tool-use datasets serves to further enhance the capabilities of these agents [ZLZ⁺24]. Despite these advancements, there is, to our knowledge, an underlying assumption that persists in the design of ~~leading AI agents and the instruction datasets they are trained on~~ these agents are inherently *synchronous*. This synchronous nature becomes evident through user interactions, such as watching a loading spinner when an agent employs a tool. Giving agents the ability to (1) manage multiple concurrent processes in real-time, and (2) asynchronously respond to the user as soon as any process finishes will significantly reduce perceived delay and meaningfully improve user experience.

Consider the following hypothetical scenario of a voice call with an AI travel concierge (see Fig. 1). At a glance, this interaction may seem uninteresting. However, consider that preparing a detailed travel itinerary might be a multi-step and multi-tool task that may take, say, 30 seconds to fully prepare. Thus, it is preferable for the concierge to quickly verbally reply rather than wait to respond until the itinerary is ready. Furthermore, the user asked for a second tool-use request while still waiting on the first. While humans have the ability to maintain multiple concurrent thought processes simultaneously, current frontier agents are fully serial¹. When a serial agent is in the middle of processing a user request that involves long-running tools or reasoning chains, the user has no choice but to abort the request or wait for it to finish before being able to submit a subsequent query.

¹See, for example: chatgpt.com, claude.ai, or perplexity.ai

*Core contributors.

USER: Please present detailed travel itinerary for my trip to Miami next week.
CONCIERGE: Certainly! I will prepare this for you momentarily.
USER: Also, what's the weather going to be like?
CONCIERGE: Based on current forecasts, the weather in Miami next week is expected to be warm and humid, with daytime highs around 88°F and overnight lows around 76°F. There's a chance of afternoon thunderstorms on several days, which is typical for Miami this time of year. I'd recommend packing light, breathable clothing, sunscreen, and a light rain jacket or umbrella. Would you like me to include some indoor activity options in your itinerary in case of rain?
USER: Sounds great!
CONCIERGE: Here is your itinerary, including indoor activity options...

Figure 1: Hypothetical voice call between a user and a travel agent with *asynchronous* tool use.

While frontier models are indeed highly general, and are in fact able to solve a variety of tasks zero-shot, including various forms of tool usage, we find that even with specialized prompting, frontier models struggle to operate in an asynchronous fashion under certain circumstances. For example, while specialized prompting does enable frontier LLMs to operate asynchronously to some degree, they may still get confused in certain scenarios with out-of-order messages.

Quite recently, OpenAI has released a real-time voice API² that supports asynchronous tool usage. However, few technical, architecture, or model details have been shared. To our knowledge, there is no other system supporting real-time voice and asynchronous tool usage.

Other available frontier LLMs seem to lack a precise notion of time and a perfect understanding of asynchronous messages. It is hard to address this problem today due to a lack of instruction tuning data for asynchronous tool usage. Furthermore, the machine learning systems co-design aspects of this problem requires some specific familiarity across both subfields.

In this work, we draw inspiration from the literature of concurrent programming and asynchronous computer system in order to present a conceptual framework and practical architecture for building asynchronous agents with frontier language models.

We propose and implement a novel event-driven finite-state machine architecture for executing and prompting AI agents in asynchronous tool environments. The system, herein called the execution environment, is complete with peripherals such as automatic speech recognition and text-to-speech. Our system is modular, and can work with any LLM that generates valid messages. We fine-tune both Llama 3.1 and GPT-4o to operate the execution environment

²openai.com/index/introducing-the-realttime-api/

and find compelling performance from both models. We discuss and characterize trade-offs in event-driven AI architectures, such context management via forking vs. spawning.

2 BACKGROUND & RELATED WORKS

We provide some background, with a focus on more fundamental and seminal works on the systems side (given that we adopt classically established systems paradigms) and more contemporaneous, bleeding-edge works on the generative machine learning side (given that this field is dynamic and advancing rapidly).

2.1 Asynchronous Computer Systems

2.1.1 Asynchronous Execution. Asynchronous execution, a cornerstone of our proposed framework for AI agents in tool environments, has its roots in the seminal work of Dijkstra (1965) on cooperating sequential processes [Dij02]. Hoare [Hoa78] offered a formal framework for describing and analyzing asynchronous systems, which informs our approach to designing AI agent interactions in asynchronous environments.

2.1.2 Polling v. Interrupt-Based Concurrency. In developing our conceptual framework for AI agents in asynchronous tool environments, we draw upon the long-standing debate between polling and interrupt-based approaches to concurrency. [Han73] offers insight into this trade-off in the context of operating systems. Lampson and Redell’s [LR80] work on processes and monitors in Mesa offers insights into the trade-offs impacting a system’s responsiveness to asynchronous events, such as automatic speech recognition inputs. However, perhaps the most influential and relevant prior work is the Robot Operating System (ROS) [K⁺17], which opts for an event-driven concurrency model.

2.1.3 Real-Time Systems. Real-time computing systems, and operating systems in particular, is an established branch of computer engineering that encompasses systems building for real-time timing requirements in a concurrent environment with integrated peripheral sensors and actuators. We include references to recent surveys [HVJ14, SR04, Meg05, DNJN11] and classical texts [Rog01, RDJ⁺09, LL73, SR89].

2.2 Generative AI

2.2.1 Large Action Models and Tool-Use. Over the past three years, as LLM model weights and training data grew from the gigabyte to terabyte scale, LLM use-cases expanded from simple chatbots to helpful copilots all the way to now, tool-using autonomous agents.

A recent study introduced xLAM, a series of large action models specifically designed for AI agent tasks, ranging from 1B to 8x22B parameters [ZLZ⁺24]. The xLAM models demonstrate exceptional performance across multiple agent ability benchmarks, notably outperforming GPT-4 and Claude-3 in function calling tasks, and aim to advance open-source LLMs for autonomous AI agents. APIGen, an automated pipeline for generating high-quality function-calling datasets, leverages over 3,600 APIs across 21 categories for data generation [LHZ⁺24].

2.2.2 Multi-Agent Systems. Multi-agent systems in the context of generative AI have seen significant advancements in recent years. Rare challenges and opportunities in developing AI systems

that can effectively collaborate. The literature on cooperating multi-agent AI systems is vast, but some particularly important or relevant works from the before the foundation model era include [TOKŠ17, YVV⁺22, Kra97, Jen95, DKJ18].

More recently, the advent of generative foundation models has ushered in a new era of AI coordination amongst LLM-based agents. Development of platforms like AutoGen [WBZ⁺23] provides practical tools for implementing and experimenting with multi-agent AI systems, facilitating research and applications in this rapidly evolving field.

Diversity Empowered Intelligence (DEI) is framework that leverages the diverse expertise of multiple software engineering agents to enhance problem-solving capabilities. Experimental results demonstrated that a DEI-guided committee of agents significantly outperformed individual agents [ZYL⁺24].

AgentLite is lightweight open-source library designed to simplify the development and evaluation of LLM-based agent systems [LYZ⁺24]. AgentLite offers a task-oriented framework that enhances agents’ ability to break down tasks and facilitates multi-agent system development, providing researchers with a user-friendly platform for innovating LLM agent reasoning strategies and architectures.

Unlike prior work, our work is far more focused on the asynchronous and real-time aspects of LLM-based agents rather than the cooperation and coordination aspects of multi-agent systems. Many of the ideas in the prior multi-agent systems literature are in fact quite complimentary and could be translated to the asynchronous setting. Other relevant works on multi-agent generative systems include [GCW⁺24, LZL⁺23, GHL⁺24, TN23, Don24, ZYB⁺24].

2.2.3 Speech Models. Speech models have undergone remarkable advancements in recent years. The introduction of Whisper [RKX⁺23] marked a significant milestone in automatic speech recognition (ASR), demonstrating robust performance across multiple languages and accents.

In the domain of text-to-speech (TTS), models like VALL-E 2 [CLZ⁺24] have shown impressive capabilities in voice cloning and style transfer, requiring only short audio samples. Voicebox [LVS⁺24] has demonstrated state-of-the-art performance in various speech generation tasks, including noise removal, content editing, and cross-lingual style transfer. NaturalSpeech [TCL⁺24] approaches human-level quality in speech synthesis.

These advancements are paving the way for more natural and versatile speech interfaces in AI systems, with potential applications ranging from virtual assistants to accessibility tools.

2.2.4 Spoken Dialogue Systems. Spoken dialogue systems have seen significant advancements in recent years, particularly in the areas of dialogue state tracking and dialogue management, due to rapid development in natural language processing powered by LLMs. [Wah23, GKD⁺24, Jan23] provide details on recent developments in this subfield.

3 REAL-TIME AGENTS

The proposed framework for real-time agents combines an asynchronous execution environment with a markup language prompting specification. Similar to a software-hardware divide, as long as

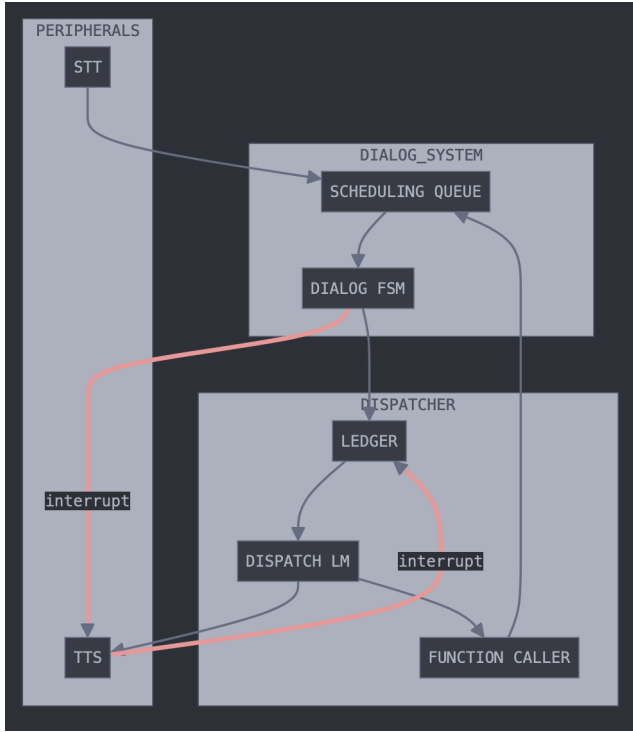


Figure 2: Architecture diagram for the asynchronous execution environment with voice peripherals

the LLM produces valid generations according to the specification, the environment will asynchronously enqueue function calls³ and handle chat interactions through speech-to-text and text-to-speech peripherals.

The asynchronous execution environment is, at its core, an event-driven finite state machine [Har87], augmented with priority scheduling (via priority queue) [MJ60]. We refer to this core component of the execution environment as the *dialog system*, comprised of the *dialog FSM* and the *scheduling queue*. The LLM generation, context management, and function calling are handled by the *dispatcher*, which is comprised of the *dispatch language model* and the *ledger*, which acts as the single-source-of-truth for the dispatch LM’s context. Messages are appended to the ledger atomically, generally as a consequence of the dialog FSM processing the scheduling queue, but there are exceptions, such as for handling user interruptions. Refer to Fig. 2 for a schematic of the execution environment with speech-to-text (STT) and text-to-speech (TTS) I/O. While real-time voice agents are an important motivation, they are not the only use case for real-time asynchronous tool usage, so the STT and TTS messages could be replaced with input and output text streams.

3.1 Architecture

3.1.1 Clock Awareness. One important aspect of our paradigm arises from the usage of timestamps in messages, clock update messages at discrete time intervals (for example, every 5 seconds)

³We use the terms functions and tools interchangeably.

coupled with a notion of clock awareness present in the LLM (due to specialized fine-tuning or prompting).

3.1.2 Parallel Thought Processes: Fork vs. Spawn Semantics. We propose a technical definition of a *parallel thought process*. Analogous to a parallel programmatic process in a traditional software sense, parallel thought process is an concurrent instance of the asynchronous execution environment with parent-child semantics. The child’s input stream is populated by function calls from the parent and the child’s output stream populates the function responses to the parent. Concretely, parallel thought processes are created via either fork or spawn calls. For a fork call, the parent initializes the child’s ledger with a copy of its own and appends a new message containing further instructions for the child. For a spawn call, the parent initializes the child with a new ledger and populates the first message containing the child’s instructions. It is up to the parent thought processes to determine if a fork or a spawn call is more appropriate on a case-by-case basis, given that there are clear trade-offs for each type of parallel thought process. Forking uses more context in the child and therefore could be more expensive while also including potentially unnecessary or distracting messages, so it should only be used if the child needs a full view into the parent’s context in order to achieve its goal. By default, spawning is probably preferable for most cases, since the parent can usually summarize the relevant details into the child’s instructions. By default, the dispatch LM could be the same as the parent, but in principle, nothing prevents the parent from prescribing a different dispatch LM (as long as it’s fine-tuned or prompted to correctly handle prompt template as expected by the environment). For both fork and spawn, we have a third reserved tool, *kill*, that the dispatching LM can use to interrupt and terminate parallel thought process. As is implied by these function call semantics, there is the possibility of recursive creation of parallel thought processes, which could be used to dynamically organize multi-agent hierarchies at runtime. Of course, custom-built tool calls can also serve this purpose.

3.1.3 Event Processing. The dialog system implements an event-driven FSM with priority scheduling. An event will contain a priority level and (potentially) cause a state transition to occur. Some events also contain messages to be appended to the ledger. Events can be produced by:

- (1) The STT (or input peripheral), when the user begins speaking or finishes speaking
- (2) The dispatcher, when the dispatch LM begins or finishes generation
- (3) The TTS (or output peripheral), when the output stream begins and finishes emitting
- (4) The function caller, when a tool-use request is sent or a response is received

In order to ensure that the dialog FSM state variable accurately reflects the overall system, state transitions push from the dispatcher and the peripherals should essentially always have the minimum possible priority, $-\infty$, in order to ensure they are processed instantly. Alternatively, the dispatcher and the peripherals can use a locking mechanism to atomically update the state variable when appropriate (essentially skipping the scheduling queue altogether). Function call responses, on the other hand, should never use such a

priority, and instead should use a developer defined priority, always going through the scheduling queue. push events to the scheduling queue based on on internal processing or state change. As a concrete example, the execution environment must ensure that the TTS is halted if the interrupt event is pushed by the STT subsystem because the user starts to speak.

Event	Priority	Message	State
generate_done	$-\infty$	No	Idle
emit	$-\infty$	No	Emitting
emit_done	$-\infty$	No	Idle
interrupt	$-\infty$	Yes	Listening
tool_response_received	p	Yes	Generating
user_chat	-1	Yes	Generating
tool_request_sent	$-\infty$	Yes	Idle
time_passage	1	Yes	Generating

Table 1: Events. The Priority column is the priority level in the scheduling queue (p denotes a tool-specific variable-defined priority level). The Message column is *Yes* if the event contains a message to be appended to Ledger and *No* otherwise. The State column contains the next state for the FSM to transition into. The FSM has four states: *idle*, *listening*, *generating*, and *emitting*. There exist certain environment conditions under which the emit done event would actually transition to a generating state. See Section 3.3.3 for more details.

3.2 Peripherals

We use the open-source Pipecat⁴ framework to integrate our LLM with STT and TTS peripherals. This framework enables real-time voice recognition and synthesis, and helps handle user interruptions. It does not support asynchronous tools, however, which we implement in our system. LLM generations are streamed into a sentence aggregator and then processed phrase-by-phrase by the TTS. We observe end-to-end latency of <300ms, which is enough to mimic human conversation.

3.2.1 Speech-to-Text. There are various suitable choices for STT services, both among the open-source packages and APIs. We found Deepgram⁵ to be a good API option. We also obtained low-latency STT with an optimized implementation of Whisper Turbo⁶.

3.2.2 Text-to-Speech. We found that TTS had a slightly higher variation in latency and quality among leading open-source packages and APIs. We preferred the Sonic API⁷ for its speed and quality.

3.3 Execution

OpenAI introduced *chat markup language* (ChatML)⁸ as the original prompting template for fine-tuned chat models. ChatML originated as a specific markup language that implement an abstract data type (ADT), but, interestingly, it was the ADT implied by ChatML that has actually caught on as the universal standard interface to chat models for developers across all major APIs. Today, the ChatML interface enables developers to more easily manipulate LLM context without resorting to boilerplate string manipulations.

We propose an extension to ChatML for asynchronous real-time agents. However, beyond just being an interface, we actually view this as closer to an *instruction set* because it provides an abstraction layer between the chat model and the asynchronous execution environment. Any chat model that generates valid (asynchronous) ChatML, should, in principle, be able to run on any asynchronous execution environment that implements the specifications.

3.3.1 Prompt Template and Context Management. We review the standard practice for synchronous chat model context, here denoted by C . Precisely, context C is a list of length ℓ of message dictionaries, m_i , with each message dictionary containing role ($role_i$) and content ($content_i$) fields.

$$C = ((role_1, content_1), \dots, (role_\ell, content_\ell))$$

where $role_i \in \{\text{system}, \text{assistant}, \text{user}\}$ and $content_i \in \Sigma^*$ where Σ denotes the token vocabulary for the language model (including the empty string). We do not impose any restrictions about the number or order of the roles in the context. For example, we allow back-to-back user messages.

In the asynchronous case, the ledger, L is similarly a list of messages, $(role_i, content_i)$. We include a set of tools, denoted \mathcal{T} , modeled via *tool-use function* $f : \text{JSON} \rightarrow \Sigma^*$, which inputs valid JSON, selects and executes the corresponding tool in \mathcal{T} with the corresponding arguments, and outputs a response string. We also include certain *reserved tools*, including fork, and spawn and kill. We include a new, fourth role, notification that is managed by the asynchronous execution environment. It will also be useful to define a set of hash strings \mathbb{H} .

If $role_i = \text{user}$

$$content_i = (\text{timestamp}_i, \text{chat}_i)$$

where $\text{timestamp}_i \in \mathbb{N}$ (or some other suitable timestamp format) and $\text{chat}_i \in \Sigma^*$.

If $role_i = \text{assistant}$

$$content_i = (\text{thought}_i, \text{function}_i, \text{chat}_i)$$

where $\text{thought}_i \in \Sigma^*$ and $\text{function}_i \in \text{List}[\text{JSON}]$ is a list of valid JSONs.

If $role_i = \text{notification}$

$$content_i = (\text{source}_i, \text{timestamp}_i, \text{data}_i)$$

The source field where $\text{source}_i \in \mathcal{T} \times \mathbb{H}$ and $\text{thought}_i \in \Sigma^*$. We do not change the content or format for system messages.

Like with synchronous chat markup language, the choice of special tokens and details of the prompting template to serialize and deserialize the abstraction is specific to the language model's vocabulary.

Frontier models now offer context lengths upwards of 128k tokens, which can enable voice conversations with tool calls lasting up to 30 minutes or more. Context length and memory management beyond this is an interesting topic which we do not delve into here.

3.3.2 Instruction Set. We describe how the asynchronous execution environment should handle messages. Like an instruction-tuned chat model, it is the dispatch LM's burden to ensure it generates valid messages with respect to the instruction set.

⁴github.com/pipecat-ai/pipecat

⁵deepgram.com

⁶huggingface.co/openai/whisper-large-v3-turbo

⁷cartesia.ai/sonic

⁸github.com/openai/openai-python/blob/release-v0.28.0/chatml.md

The dispatch LM is the only component of the system that generates assistant messages. For the thought field the environment does nothing. This is just an LLM scratchpad. For the chat field, the environment is responsible for initializing a real-time output stream to the user (either via a text-to-speech peripheral or a token stream directly). Finally, the environment should parse the function field and asynchronously execute the requests. Note that, in our system, token generation and TTS emitting happen concurrently, and it is technically possible for TTS emitting to finish before the generation of the assistant message. For example, consider the case in which an assistant message begins with a chat and then includes a thought. While the message as an abstract data type is a dictionary, when implemented, it streams in serially and we process it in real-time as a stream. If a chat streams in, the TTS component will start emitting sentence-by-sentence, meaning the TTS will run in a delay, but it is possible that the TTS could finish emitting the chat before the LLM generation finishes the subsequent thought. We omit the details of this edge case in Table 1 and Section 3.1, but it is important to handle this case because the dialog FSM state should revert back to generating rather than transition to idle in such a situation.

Algorithm 1: Handling Events
Input: Queue Q , Environment State E
Output: Updated Ledger L , Updated State E
 $p \leftarrow Q.\text{top.priority}$
 $\text{run} \leftarrow E = \text{idle or}$
 $(E = \text{generating and } p \leq 1) \text{ or}$
 $(E = \text{emitting and } p < 1)$
if run then
 $\text{event} \leftarrow Q.\text{pop}()$
 $L.\text{append}(\text{event.message})$ // No-op if message is empty

 $E \leftarrow \text{event.state}$
else if $E = \text{listening}$ then
 pass // optionally, we could interrupt the user if $p < -1$
return L, E

Upon the successful start of the function call (for example, asynchronously sending an HTTPS request), the execution environment should append a *request-sent* notification message. This message should include the name of the tool used, a request ID, as well as potentially other metadata such as the arguments. For example:

Message: j
 $\text{role} = \text{notification}$
 $\text{source} = \text{system}$
 $\text{data} = \text{"Request sent for: search. ID: 0abd754d495."}$

Upon the completion of the j -th function call from the i -th message, the execution environment should append a *response-received* notification message with $\text{data} \leftarrow f(\text{function}_{ij})$. For example:

Message: $j + 1$
 $\text{role} = \text{notification}$
 $\text{source} = (\text{search}, \text{"0abd754d495"})$
 $\text{data} = \text{"Here are your results..."}$

The user messages are essentially unchanged and still represent direct input from the user. The environment adds timestamps.

3.3.3 Interruption Handling. At any given moment in time, the FSM is in one of four states with respect to the execution environment: *idle*, *generating*, *emitting*, or *listening*. The implementation of the execution environment as a whole is responsible for ensure the FSM state is accurately reflected in the true state of the system.

For example, if the TTS is streaming output (either voice or text) to the user, then the FSM is emitting. Otherwise, if it is generating tokens but not emitting, then it is generating. If the user is in the process of creating input (for example, speaking), then the FSM is listening. Else, the FSM is said to be idle.

Interruptions are a first-class feature in an asynchronous agent. We explicitly include interruptions as part of the proposed instruction set. The scheduling queue allows for the environment to enforce atomic updates to the ledger despite the concurrency. Every 5 seconds, the system queues up a time passage notification message.

All messages have a priority⁹. As example defaults, user messages are priority -1 and assistant messages are priority 1 (although this can be configured on a per-deployment basis).

The key difference in how interruptions are handled in the *generating* and *emitting* states is that for a tie in priority level, the interrupt occurs if the dispatch LM generating but does not if it is emitting.

Tool definitions should include a priority, but the default priority for request-sent and response-received messages is 1.

The execution environment is responsible for correctly book-keeping the ledger during interruptions. Consider the following example.

Message: l
 $\text{role} = \text{assistant}$
 $\text{chat} = \text{"Blah blah blah <|interrupt|>"}$
Message: $l + 1$
 $\text{role} = \text{notification}$
 $\text{source} = \text{system}$
 $\text{data} = \text{"Assistant interrupted due to user speaking"}$
¹⁰**Message: $l + 2$**
 $\text{role} = \text{user}$
 $\text{chat} = \text{"I am interrupting you."}$

We add a special interruption token, as depicted above. Note that, due to the pipeline nature of the asynchronous execution environment, there is a time delay between token generation and the output stream (for example, via a TTS module). The system is responsible for making sure the number of *blah* accurately reflects how many the assistant said before interruption. In this case, the LLM may have generated a token stream of "Blah blah blah blah" while the TTS has only managed to emit "Blah blah blah". In this case, the environment is responsible for reconciliation between these two streams and should only update the ledger to reflect the actual output emitted to the user. This issue is only present in the chat component of the LLM generation, since thought streams do not trigger downstream post-processing and therefore can be interrupted naively. For function call streams, the environment

⁹Inherited from the priority of the event containing said message

¹⁰Note that message $l + 1$ is posted to the ledger immediately when the user starts speaking, whereas message $l + 2$ is posted once the user is done.

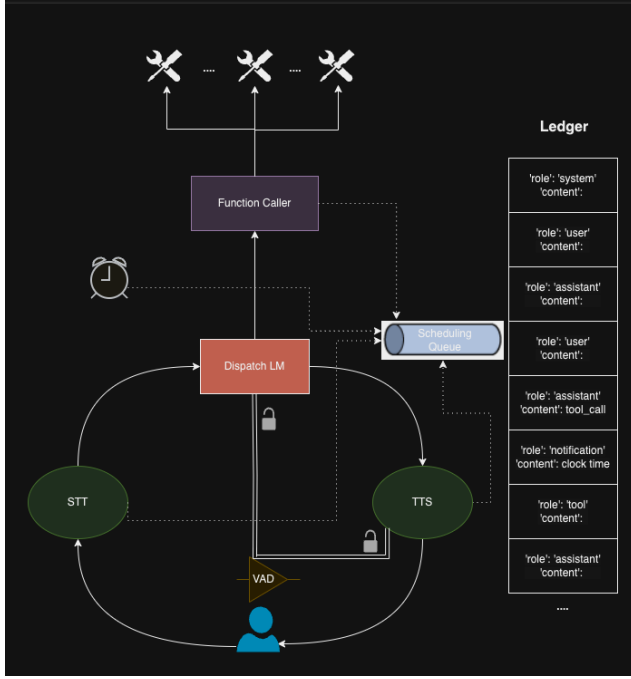


Figure 3: Implementation and control flow diagram for the asynchronous execution environment

could choose to either atomically include a function call only if it was generated to completion, or simply interrupt a partial function call string. In either case, the dispatch LM should rely on a request-sent message to indicate the completion of the function call request. Below, Fig. 3 offers another perspective on the system implementation, emphasizing the ledger and messages, the control and information flow through various system components and the voice activity detection (we implemented a simple volume based cutoff, see [SKS99, MH09, CKM06, ZW12, HM13] for more details and more sophisticated approaches).

4 DISCUSSION

Results. In the supplementary materials, we share some sample recordings of voice calls with both the fine-tuned Llama 3.1 and GPT-4o as dispatch LMs. Qualitatively, we find that fine-tuning helps ensure the dispatch LM consistently generates valid ledger messages according to the prescribed instruction set.

Pipelined vs. Multi-modal Systems. Multi-modal foundation models are an active and promising area of research and development, and are highly relevant to real-time agents, particularly for voice I/O. A foundation model with both text and speech modalities could exhibit tighter integration with peripherals and possibly even further reduce latency or improve speech quality.

In our implementation, we opted for a pipelined system, in which we had independent neural networks for the STT, Dispatch LM, and TTS, which text as the I/O medium between the subnets. However, one could have also used a multi-modal foundation model (as demonstrated by OpenAI). Our architecture is well-suited for

such a foundation model, with the exception that the STT and TTS components would not include a neural component, but rather, would simply include a speech tokenizer and vocoder component, respectively. In this case, speech tokens would be the I/O medium between these components.

Reasoning for Time-Constrained Tasks. One intriguing possibility that is enabled after endowing the agent with a sense of clock awareness is the ability for the user to request a *time-constrained* task. For example: *Complete a research report on state-of-the-art text-to-speech models within the next 10 hours.* Within the proposed framework, the agent’s sense of clock awareness enables it to coordinate long-running chain-of-thought (or other multi-step reasoning methods) with multiple iterations of tool-use across, potentially, multiple other task-specific agents while keeping the entire process on a reasonable schedule based on the allotted time limit.

Exploring specific prompt templates and fine-tuning strategies for powering this kind of clock-aware reasoning with time constraints is a fertile topic for future work.

Conclusion and Future Work. This work introduces an architecture for real-time, asynchronous AI agents capable of fluid, real-time interactions. By implementing an event-driven finite-state machine with asynchronous tool usage and parallel thought processes, we enable more natural and responsive AI interactions, particularly in voice-based applications.

Broadly speaking, we envision a future in which agent LMs are fine-tuned to for precise instruction set specifications which can be executed in any environment supporting said instruction set. Future work should explore integration with multi-modal LMs and seamless integration with multi-agent systems and long-running task-specific agents.

REFERENCES

- [CKM06] Joon-Hyuk Chang, Nam Soo Kim, and Sanjit K Mitra. Voice activity detection based on multiple statistical models. *IEEE Transactions on Signal Processing*, 54(6):1965–1976, 2006.
- [CLZ⁺24] Sanyuan Chen, Shujie Liu, Long Zhou, Yanqing Liu, Xu Tan, Jinyu Li, Sheng Zhao, Yao Qian, and Furu Wei. Vall-e 2: Neural codec language models are human parity zero-shot text to speech synthesizers. *arXiv preprint arXiv:2406.05370*, 2024.
- [Dij02] Edsger W Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 65–138. Springer, 2002.
- [DKJ18] Ali Dorri, Salil S Kanhere, and Raja Jurdak. Multi-agent systems: A survey. *Ieee Access*, 6:28573–28593, 2018.
- [DNJN11] Mark L Darby, Michael Nikolaou, James Jones, and Doug Nicholson. Rto: An overview and assessment of current practice. *Journal of Process control*, 21(6):874–884, 2011.
- [Don24] Yihan Dong. The multi-agent system based on llm for online discussions. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*, pages 2731–2733, 2024.
- [GCW⁺24] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges, 2024.
- [GHL⁺24] Xudong Guo, Kaixuan Huang, Jiale Liu, Wenhui Fan, Natalia Vélez, Qingyun Wu, Huazheng Wang, Thomas L. Griffiths, and Mengdi Wang. Embodied llm agents learn to cooperate in organized teams, 2024.
- [GKD⁺24] Chulaka Gunasekara, Seokhwan Kim, Luis Fernando D’Haro, Abhinav Rastogi, Yun-Nung Chen, Mihail Eric, Behnam Hedayatnia, Karthik Gopalakrishnan, Yang Liu, Chao-Wei Huang, et al. Overview of the ninth dialog system technology challenge: Dstc9. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 2024.
- [Han73] Per Brinch Hansen. *Operating system principles*. Prentice-Hall, Inc., 1973.
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

- [HM13] Thad Hughes and Keir Mierle. Recurrent neural networks for voice activity detection. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 7378–7382. IEEE, 2013.
- [Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
- [HVJ14] Prasanna Hambarde, Rachit Varma, and Shivani Jha. The survey of real time operating system: Rtos. In *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, pages 34–39, 2014.
- [Jan23] Dietmar Jannach. Evaluating conversational recommender systems: A landscape of research. *Artificial Intelligence Review*, 56(3):2365–2400, 2023.
- [Jen95] Nicholas R Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial intelligence*, 75(2):195–240, 1995.
- [K⁺17] Anis Koubaa et al. *Robot Operating System (ROS)*, volume 1. Springer, 2017.
- [Kra97] Sarit Kraus. Negotiation and cooperation in multi-agent environments. *Artificial intelligence*, 94(1-2):79–97, 1997.
- [LHZ⁺24] Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets, 2024.
- [LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.
- [LR80] Butler W Lampson and David D Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, 1980.
- [LVS⁺24] Matthew Le, Apoorv Vyas, Bowen Shi, Brian Karrer, Leda Sari, Rashel Moritz, Mary Williamson, Vimal Manohar, Yossi Adi, Jay Mahadeokar, et al. Voicebox: Text-guided multilingual universal speech generation at scale. *Advances in neural information processing systems*, 36, 2024.
- [LYZ⁺24] Zhiwei Liu, Weiran Yao, Jianguo Zhang, Liangwei Yang, Zuxin Liu, Juntao Tan, Prafulla K. Choubey, Tian Lan, Jason Wu, Huan Wang, Shelby Heinecke, Caiming Xiong, and Silvio Savarese. Agentlite: A lightweight library for building and advancing task-oriented llm agent system, 2024.
- [LZL⁺23] Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. Dynamic llm-agent network: An llm-agent collaboration framework with agent team optimization, 2023.
- [Meg05] S Baskiyar N Meghanathan. A survey of contemporary real-time operating systems. *Informatica*, 29(2), 2005.
- [MH09] Mohammad Hossein Moattar and Mohammad Mehdi Homayounpour. A simple but efficient real-time voice activity detection algorithm. In *2009 17th European signal processing conference*, pages 2549–2553. IEEE, 2009.
- [MJ60] Rupert G Miller Jr. Priority queues. *The Annals of Mathematical Statistics*, 31(1):86–103, 1960.
- [OWJ⁺22] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.
- [PLH⁺23] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. Instruction tuning with gpt-4, 2023.
- [RDJ⁺09] Franz Rammig, Michael Ditze, Peter Janacik, Tales Heimfarth, Timo Kerstan, Simon Oberthuer, and Katharina Stahl. Basic concepts of real time operating systems. *Hardware-dependent Software: Principles and Practice*, pages 15–45, 2009.
- [RKX⁺23] 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*, pages 28492–28518. PMLR, 2023.
- [Rog01] P Rogers. "real-time systems and programming languages" by alan burns and andy wellings. *ADA USER JOURNAL*, 22(2):126–126, 2001.
- [SKS99] Jongseo Sohn, Nam Soo Kim, and Wonyong Sung. A statistical model-based voice activity detection. *IEEE signal processing letters*, 6(1):1–3, 1999.
- [SR89] John A. Stankovic and Krithi Ramamritham. The spring kernel: A new paradigm for real-time operating systems. *ACM SIGOPS Operating Systems Review*, 23(3):54–71, 1989.
- [SR04] John A Stankovic and Raj Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3):237–253, 2004.
- [TCL⁺24] Xu Tan, Jiawei Chen, Haohe Liu, Jian Cong, Chen Zhang, Yanqing Liu, Xi Wang, Yichong Leng, Yuanhao Yi, Lei He, et al. Naturspeech: End-to-end text-to-speech synthesis with human-level quality. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.
- [Tea22] Stanford AI Lab Team. On the opportunities and risks of foundation models, 2022.
- [Tea24] Meta Gen AI Team. The llama 3 herd of models, 2024.
- [TN23] Yashar Talebira and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent llm agents, 2023.
- [TOKŠ17] Alejandro Torreno, Eva Onaindia, Antonin Komenda, and Michal Štolba. Cooperative multi-agent planning: A survey. *ACM Computing Surveys (CSUR)*, 50(6):1–32, 2017.
- [Wah23] Wolfgang Wahlster. Understanding computational dialogue understanding. *Philosophical Transactions of the Royal Society A*, 381(2251):20220049, 2023.
- [WBZ⁺23] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Auto-gen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.
- [YVV⁺22] Chao Yu, Akash Velu, Eugene Vinitzky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems*, 35:24611–24624, 2022.
- [ZLZ⁺24] Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, Zhiwei Liu, Yihao Feng, Tulika Awalganekar, Rithesh Murthy, Eric Hu, Zeyuan Chen, Ran Xu, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. xlam: A family of large action models to empower ai agent systems, 2024.
- [ZW12] Xiao-Lei Zhang and Ji Wu. Deep belief networks based voice activity detection. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(4):697–710, 2012.
- [ZYB⁺24] Yang Zhang, Shixin Yang, Chenjia Bai, Fei Wu, Xiu Li, Zhen Wang, and Xuelong Li. Towards efficient llm grounding for embodied multi-agent collaboration, 2024.
- [ZYL⁺24] Kexun Zhang, Weiran Yao, Zuxin Liu, Yihao Feng, Zhiwei Liu, Rithesh Murthy, Tian Lan, Lei Li, Renze Lou, Jiacheng Xu, Bo Pang, Yingbo Zhou, Shelby Heinecke, Silvio Savarese, Huan Wang, and Caiming Xiong. Diversity empowers intelligence: Integrating expertise of software engineering agents, 2024.