Simple Action Model: Enabling LLM to Sequential Function Calling Tool Chain

Rajat Sandeep Sen
Department of AI&DS
SJCET Palai
Kottayam, Kerala
rajatsandeepsen2025@ai.sjcetpalai.ac.in

Amalkrishna M
Department of AI&DS
SJCET Palai
Kottayam, Kerala
amalkrishna28032003@gmail.com

Prithviraj R
Department of AI&DS
SJCET Palai
Kottayam, Kerala
prithvirajr0306@gmail.com

Sharon Prashant Jose
Department of AI&DS
SJCET Palai
Kottayam, Kerala
sharonprashantjose2025@ai.sjcetpalai.ac.in

Neena Joseph
Department of AI&DS
SJCET Palai
Kottayam, Kerala
nestofneena@gmail.com

Renjith Thomas
Department of AI&DS
SJCET Palai
Kottayam, Kerala
renampatt@gmail.com
ORCID: https://orcid.org/0000-0002-1038-9596

2024 International Conference on Advancement in Renewable Energy and Intelligent Systems (AREIS) | 979-8-3503-8723-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/AREIS62559.2024.10893677

***Abstract* – Today, LLMs are everywhere. It is making human internet life a lot easier than ever. Every day new sophisticated models are released. But these models are not good enough to become personal assistants like the Jarvis from the Sci-fi movie IronMan. This paper proposes a way to enable any LLM to execute complex requirements in real-world applications. By leveraging state-of-the-art Large Language models, we can create a simple action model that can understand the environment around them. Eventually, these models can help or assist humans in real-time applications. The Sequential Function Calling Tool Chain System aims to bridge the gap between human language understanding and computer programming.**

Keywords* – *Large language model, Action model, OpenAPI format, and Function Calling Tools

I. Introduction

Nowadays, a world without AI is impossible to imagine. New models will come every day, and new architecture will improve its performance. Multi-modality or a Mixture of Models is not enough to become a personal assistant. Graph-based models take up a lot of computation in the case of complex function calling. Simple Question, " What if the one good model with one request is enough to execute multiple functions sequentially?".

The integration of OpenAPI schema with LLM can make a decent action model that can understand the API specs of that specific application. The large language model will take prompts and decide what to do. From the programming side, we'll parse the information and execute it. A simple action model is done.

By implementing a custom JSON parser alongside an OpenAPI schema-type system it is possible to utilize one model as main decision making inside an application. This technology works faster compared to other solutions but requires a much more intelligent language model.

This helps to provide stable and intelligent AI assistants in any applications that uphold the integrity and standards of API formats.This research paper also leads to the exploration of Multiple and Sequential Function calling toolchain that utilizes a State-of-the-Art Language model to enhance the security, time efficiency, and flexibility of building AI applications.

The purpose of this paper is to explore and improve existing solutions of function calling tools available in the open-source software community. This paper also aims to provide a new approach to prompting the LLMs without fine-tuning.

II. Literature Survey

These literature reviews provide knowledge about the existing research done by various scholars and open-source developers.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, Graham Neubig, PAL: Program-aided Language Models, Jan 2023 [1], They had introduced a new architecture of LLMs wrapper that are trained on both interpreted programming language and natural languages. These system are utilized with ability to run code directly at runtime. Tatsuro Inaba, Hirokazu Kiyomaru, Fei Cheng, Sadao Kurohash, MultiTool-CoT: GPT-3 Can Use Multiple External Tools with Chain of Thought Prompting (2023) [2], They had introduced a new architecture of LLMs that can use multiple external tools with Chain of Thought Prompting. 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, Maosong Sun, TOOLLLM: FACILITATING LARGE LANGUAGE MODELS TO MASTER 16000+ REAL-WORLD APIS 2023 [3], They had introduced a new architecture of LLMs that can master 16000+ real-world APIs. Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, Amir Gholami An LLM Compiler for Parallel Function Calling Jun 2024 [4], They had introduced a

979-8-3503-8723-0/24/$31.00 ©2024 IEEE

new architecture of LLMs that can compile parallel function calling.

III. Methodology

The approach of the Sequential Function calling tool is using the popular and newest OpenAPI schema 3.0. By fetching the schema JSON from the backend, it is converted to a type parser like Zod (npm) at build time. The prompt captured from the user is fed to LLM with proper type annotation of OpenAPI schema [7]. Any popular open-source model can be used to understand and extract object/JSON data from user's prompts. The extracted data are then loaded into a custom JSON parser that supports extra keywords from the standard JSON format [9]. The parser moves the data to OpenAPI client libraries like OpenAPI-Fetch (npm), which fetches results from the backend. The result is then passed to the next function in the chain. The process is repeated until the end of the chain [8]. But LLM is used once just to write "what to do in this environment" according to users' prompts and type definition.

A. Schema Preparation

The preparation of the schema at build time utilizes popular libraries like OpenAPI-TS (npm) and OpenAPI-Zod (npm). It'll generate types schema for use in few-shot learning of LLM and zod schema for parsing the input and return data to and from LLM [11]. Feeding the LLM with raw OpenAPI schema is a waste of tokens and context window, that's why it is a crucial step to create a simpler version of the schema. This process involves collecting open API schema JSON (by fetching from the application backend server), parsing and preprocessing, and then structuring it in a way that is useful for teaching the language model.

The recorded schema was obtained from backend frameworks that support swagger-ui or other libraries that generate OpenAPI. Simplified parser schema and type are written to a coding file (here TypeScript programming language is used) and saved into a folder. Later this obtained schema was resized to smaller chunks if the schema is larger than the model's context window [5]. The graph-based structure is used to label and store nodes. This graph data structure will help to query through large schemas faster in runtime [6].

B. Prompt Preparation

The LLM is always instructed to return a response in JSON tags, making the data easier to extract from a string [4]. Also, the available functions list (or schema from OpenAPI) is converted to easy, type definitions, which makes it easier to understand for the LLMs. But normally open-source LLMs are limited without examples [3]. They are trained on general knowledge, and that's why a few shots of in-context learning are needed to make it understand the environment as it is generally seen and is described in Fig 1. [10].

Also, the fine-tuning approach modelled in Fig 2, is great if the type definition and instructions are larger than the context window of the LLM used. Teaching the model with the entire OpenAPI schema and examples before deploying it to production gives better results in response and saves a lot of tokens while prompting each time [7].

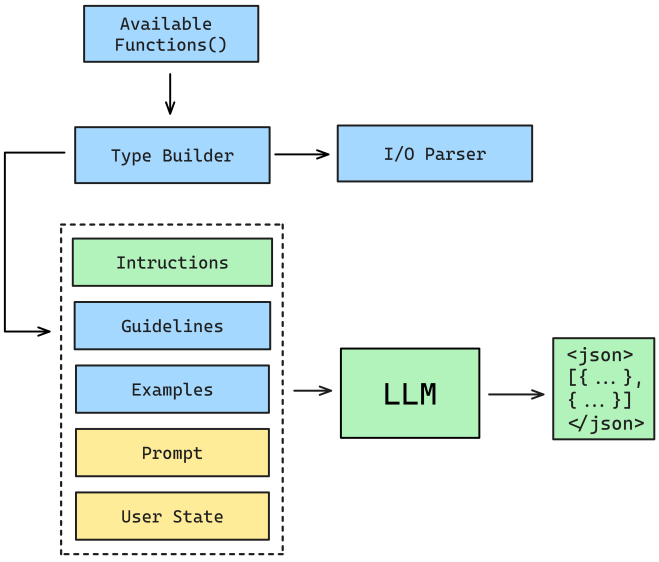

Fig. 1. Block Diagram for Normal In-context Learning

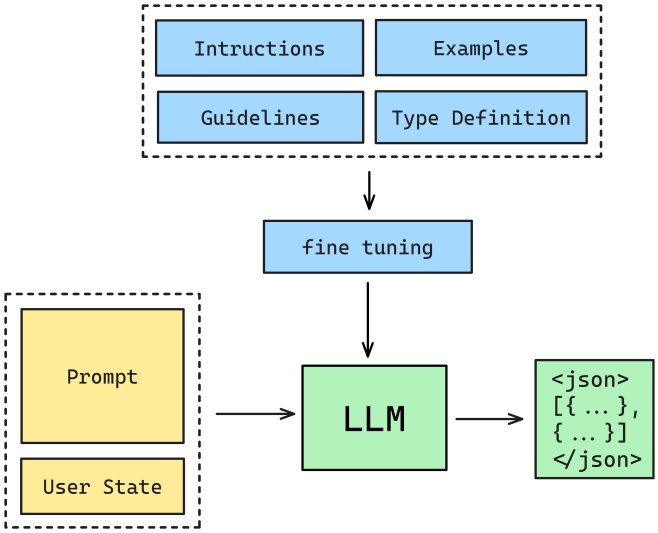

Fig. 2. Block Diagram for Fine tuned learning

C. Server Architecture

Fig 3 displays the Architecture of the Action model where initially, the developer specifies the backend URL and OpenAPI schema URL in the configuration file. It fetches the schema from the backend applications and prepares the schema at build time [2]. The server is a standalone backend that listens to end-user prompts. The server takes the user prompt, types the definition of the OpenAPI schema, and sends it to LLM. The developer also needs to mention the provider and model in the configuration file. Then the LLM returns with some JSON data wrapped with JSON tags. The data shows "what to run", "where to fetch" and "what are the parameters

and request body" [9]. Then it parses the JSON data and sends it to the Action engine. The engine takes care of function calling and chaining. Finally, the result is sent back to the user in JSON format (or stream the UI if the server is built with meta-frameworks like Next.js or Nuxt.js).

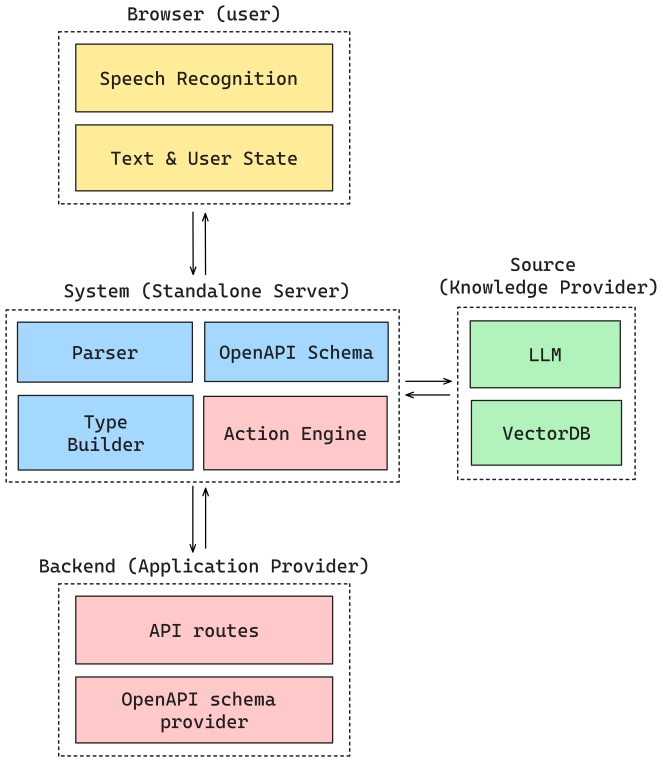

Fig. 3. Server Architecture with Action Model

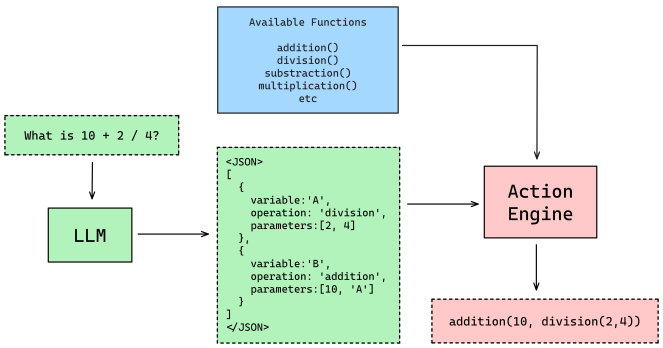

Fig. 4. Simplified Maths problem done with Function Calling.

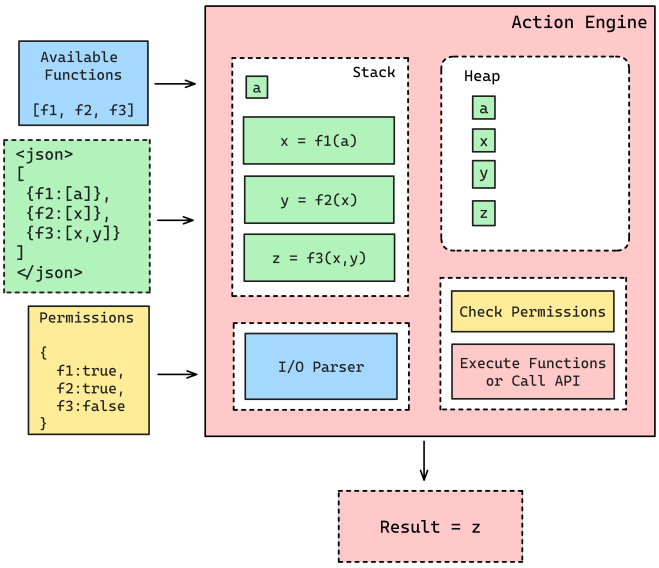

Fig. 5. Block Diagram of Action Engine Architecture

D. Action Engine Architecture

As shown in Fig 5, The Action engine architecture follows the same pattern as the build time CLI (CLI that converts the OpenAPI schema to type definitions) [5]. An action engine is not an AI system. Currently, it is a simple JSON parser that can understand the JSON data from LLM. It turns the normal Large Language Model into a Simple Action Model, that enables function calling and chaining [6].

Let's consider a simple application for mathematical calculations, the user asked "What is 10 + 2 / 4" and the Action engine had access to all types of mathematical functions. So the LLM returns instructions in JSON tag and the Action engine will execute the function in the order of the instructions [8]. The result is then sent back to the user. This whole process is described in Fig 4.

The new unknown variable is generated and used at runtime [12]. The type structure of return JSON can be customized. Here this JSON is enough to represent the environment. JSON is parsed and validated by the engine and array to move to the execution stack, and each function is called sequentially, where the result of the previous execution is stored in the heap at runtime [10].

Two types of instructions can be evaluated. Single: parse the JSON, execute the function, done. But for multiple sequential instructions, careful validations and parsing are required. The first execution of the function requires a complete set of parameters [4]. Always store the executed data and function parameters in the heap. If other instructions require data from a previous execution, search in the input params list in a stack or search heap [7]. If data is not found, return an error (and let the server handle that part).

To ensure consistency and compatibility, the input data parameters and output results from each execution are also validated before passing down to the next instructions [15]. This standardization allows for seamless processing and execution of function calling, enabling the system to effectively detect any instances of an error made by the language model [14].

The permission is the list of functions that are allowed to run without users' concerns [6]. If a function is true, then execute it without asking for permission. If not, then return the current result stack of execution to the user and wait until the user permits. This feature is crucial while building highly secure

and user data-dependent applications [13]. Users don't want any LLM to make decisions and execute some functionalities on its own without their permission [9].

E. Utilizing Vector Storage

Retrieval Augmented Generation (RAG) is a popular technique used alongside large language models (LLMs) to improve its performance by including external knowledge sources [5]. Mostly here we can utilize it with the help of vector databases.

OpenAPI schema format supports descriptions and examples. These descriptions can be stored in a vector database alongside the metadata of each API route or component [11]. If the application OpenAPI schema is larger than the context window capacity of the LLM, then utilization of vector databases can still reduce the number of requests to LLMs. Query the vector database with user prompt and get the most probable API route or component [8]. Then combine the results with user prompt to get a response from LLM. This approach is faster and more efficient than querying the LLMs with large amounts of OpenAPI schema [12].

RAG-based Action models are much more efficient when it comes to usage of tokens while querying. The fewer tokens used, the faster the model responds [7].

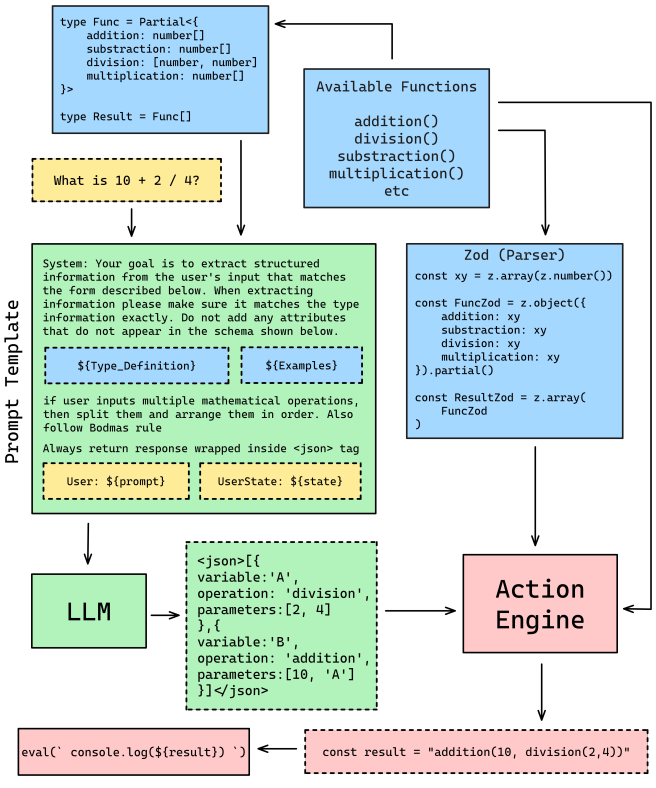

Fig. 6. Detailed Block Diagram of Maths problem

F. Utilizing Runtime Validations

Action models require strict parsing and validations with OpenAPI schema [10]. LLMs hallucinate sometimes, making them untrustworthy to take decisions. That's why it is important to validate the input and output data from fetching the API.

But what if the LLM can just go around the world wide web and fetch the schema of the API and validate at runtime and cache it? [2] This approach is much more futuristic than it sounds. The user prompts, LLM looks for relevant schemas from Google search and calls those APIs. While the system is looking for docs, it can build validators and parsers for the schema at runtime. Also, utilize a caching mechanism to store the built files for future use [16].

It's like Google search, but instead of querying, it'll do tasks online by itself [9]. An inevitable future of language models.

G. Utilizing Multiple Calls to LLM

Popular function calling projects like ReAct utilize multiple calls to LLM to make sure to get the best result and ensure the response is moving in the right direction [3]. This approach is also useful in Action models. If the response from LLM is not clear or not understandable, then a second call to LLM can reduce the errors.

This paper was focused on "how to sequentially function call with one query to LLM". But multiple calls to LLM can be useful in some cases. If return data from the previously executed function is not clear or understandable, or the data is so long that, without intermediate assistance, it may not reach the end goal as expected, multiple calls are an easy way to enhance the system [4].

H. Utilizing Program-aided Language Models

Program-aided Language Models (PALM) are the new architecture of LLMs that are trained on both interpreted programming language and natural languages. These models are utilized with the ability to run code directly in the system at runtime [8]. This approach is not preferred due to security vulnerabilities like code injection attacks [13]. The 'eval()' function allows executing arbitrary code passed as a string. So LLM might execute malicious codes [7].

Extra runtime linting features can avoid these vulnerabilities [14]. PAL gives the LLM model extra access to run the code directly into the environment [9]. This approach is suitable for building OpenAPI to function calling toolchain system [6].

IV. Implementation and Result

A. Implementation

From a developer perspective, the implementation of the Sequential Function Calling Tool Chain System involves the following steps:

- Preparation of the OpenAPI schema: The developer fetches the OpenAPI schema from the backend application and prepares the schema at build time. The schema is then converted to type definitions and saved in a folder.

- Preparation of the prompt template: The developer prepares the prompt for the LLM, ensuring that the LLM response

is in the correct format. Prompt template includes the necessary type definitions, instructions, and examples as few-shot learning.

- Server architecture: The developer must turn this into a standalone backend server that listens to end users prompts. The server takes the user prompt, combine it into template and sends to LLM. The LLM response is then parsed and sent to the Action engine. Action engine in the next step requires a fetching client library like axios (npm) or tRPC (npm). The result is then sent back to the user in JSON format or streamed as Generative UI (use Next.js or Nuxt.js for server side rending components).

- Action engine architecture: The Action engine is a JSON parser that understands the JSON data from the LLM, use kor (pip) or ai-SDK (npm). It turns the LLM into a Simple Action Model that enables function calling and chaining. The engine executes the functions in the order of the instructions and sends the result back to the user. For implementing the engine, use AI-SDK from Vercel (npm) or popular AI library LangChain (npm, pip) as base ai agent.

B. Performance Evaluation

Popular LLM benchmarks tools have 4 common metrics to consider in turning into Action Models. They are MMLU (Massive Multitask Language Understanding), Context window size and Median (inference speed or token/second).

Better the MMLU, better the model can understand the environment. More context window, gives more access to openAPI schema. Faster the inference speed, faster the model can respond with JSON. Context window and Median are inversely proportional. These results of this evaluation are gathered in Table 1.

TABLE I
PERFORMANCE METRICS COMPARISON FOR LARGE LANGUAGE MODELS

Model	Rank	Context Window	MMLU	Median (Tokens/s)
CLAUDE-3-OPUS	100	200K	0.868	27.5
GPT-4	90	8K	0.864	17.5
LLAMA3-70B	88	8K	0.82	307.3
CLAUDE-3-SONNET	85	200K	0.79	59.4
MIXTRAL-8X22B	83	65K	0.77752	49.9
CLAUDE-3-HAIKU	78	200K	0.752	85.4
CLAUDE-INSTANT	65	100K	0.734	84.2
MIXTRAL-8X7B	68	16K	0.706	117.1
GPT-3.5-TURBO	67	16K	0.7	58.5
GPT-35-TURBO	67	16K	0.7	54
LLAMA2-70B-4096	56	4K	0.689	251.5
LLAMA-3-8B	58	8K	0.684	121.4
LLAMA3-8B-8192	58	8K	0.684	920.8
MISTRAL-7B	40	16K	0.625	102.9
LLAMA-2-13B	37	4K	0.536	115.3
LLAMA-2-7B	27	4K	0.458	204.7

The evaluation of these models are provided by ArtificialAnalysis/LLM-Performance-Leaderboard from hugging face. The evaluation is based on the following metrics: API ID, Rank (Normalized avg), CONTEXT WINDOW, MMLU, MEDIAN (Tokens/s). The evaluation is based on the performance of the models in terms of their ability to generate human-like text, understand and respond to user queries, and provide accurate and relevant information. The models are ranked based on their performance across these metrics, with higher scores indicating better performance

However, it is essential to note that increasing the parameter count does not always guarantee better performance. The model's architecture, training data, and fine-tuning processes also play crucial roles in determining its overall capabilities. Therefore, it is essential to consider these factors in conjunction with the parameter count to assess the model's performance accurately. From a perspective of Enterprise, the model has to be open source or they can't fine tune the model with their own custom dataset. Fine tuning on their specialized environment can help to gain more context window through each instance of request.

C. Comparison with State-of-the-Art Models

TABLE II
TECHNICAL COMPARISON BETWEEN EXISTING SOLUTIONS AND OUR TECHNIQUE

Library	Technique Result	Function Calling
LangChain Structural Output	Call Single Function	Prompt based
OpenAI Tool API	Returns List of Functions	Tool based
Vercel's AI SDK	Call list of functions	Tool based
Cloudflare's AI Utils	Call list of functions	Tool based
Our Technique	Call nested list of functions	Both possible

Library	JSON reconstruction	Token Usage
LangChain Structural Output	Use instructor lib	High
OpenAI Tool API	Unknown	Unknown
Vercel's AI SDK	Retry on failure	Minimal
Cloudflare's AI Utils	None	Minimal
Our Technique	None	High

Table 2 shows a Technical Comparison between the Existing Solutions present and Our System by comparing their Result, Function Calling criteria, JSON reconstruction ability and Token usage. The system was also compared with other state-of-the-art models, such as ReAct and Chain-of-Thought Prompting (COT). The system outperformed these models in terms of speed, efficiency, and accuracy. The system was also found to be more user-friendly and easier to use than other models. The system's ability to execute multiple functions sequentially with a single query to the LLM makes it a valuable tool as a wrapper around consumer applications.

D. Findings

From Fig. 6, which illustrates the usefulness of the Program-aided Language Models (PALM) combined with the OpenAPI

Action Engine. Also introducing external vector knowledge base into this architecture brings more efficiency and speed to the system. The following are the main findings:

1) High speed response: Vector knowledge base of OpenAPI type definition reduce time to searching and faster response from system
2) Less usage of token: Splitting the OpenAPI type definition into chunks and vector embedding, brings faster response with lesser prompting token size.
3) Less complexity: The system is less complicated compared to Chain-of-Thought Prompting (COT) or Tree of Thoughts (TOT) architecture. This system is much more simpler and efficient, because of single query to LLM.
4) Reduction on computation: The system is much more efficient in terms of computation. The system is much more faster and efficient than other systems.
5) Practical Deployment: We have developed a practical and easily understandable applications build on top of this system. An banking app that navigates through the API routes and help user navigate applications with prompts.

V. Conclusion

The rapid development of sophisticated models has intensified competition in two primary areas: models designed for comprehensive knowledge and question answering, and those optimized for deployment on small devices. While consumers leveraging powerful cloud services may prioritize performance over privacy, the open-source community focuses on creating personal models for small devices, emphasizing user privacy and efficiency. This research contributes to that effort by enabling LLMs to perform internet-based research and execute actions, enhancing device intelligence while safeguarding user data. The proposed system is adaptable to diverse use cases, computational constraints, and schema sizes, ensuring seamless integration into existing applications. These findings represent a foundational step in bridging theoretical advancements with practical applications, acknowledging the complexity of AI systems and the need for ongoing refinement. This work sets the stage for further exploration toward practical, privacy-conscious AI solutions.

References

[1] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, Graham Neubig (Jan 2023), *PAL: Program-aided Language Models*, arXiv/Computer Science/Computation and Language, arXiv:2211.10435v2.

[2] Tatsuro Inaba, Hirokazu Kiyomaru, Fei Cheng, Sadao Kurohashi (2023) *MultiTool-CoT: GPT-3 Can Use Multiple External Tools with Chain of Thought Prompting*, arXiv/Computer Science/Computation and Language, arXiv:2305.16896v1

[3] 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, Maosong Sun (2023) *TOOLLLM: FACILITATING LARGE LANGUAGE MODELS TO MASTER 16000+ REAL-WORLD APIS*, arXiv/Computer Science/Artificial Intelligence, arXiv:2307.16789v2 .

[4] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, Amir Gholami (Jun 2024) *An LLM Compiler for Parallel Function Calling*, arXiv/Computer Science/Computation and Language, arXiv:2312.04511v3.

[5] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, Denny Zhou (Jan 2023) *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*, arXiv/Computer Science/Computation and Language, arXiv:2201.11903v6 .

[6] Zekun Li, Baolin Peng, Pengcheng He, Michel Galley, Jianfeng Gao, Xifeng Yan (Feb 2023) *Guiding Large Language Models via Directional Stimulus Prompting* arXiv/Computer Science/Computation and Language, arXiv:2302.11520v4 .

[7] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, Ji-Rong Wen (Apr 2024)*A Survey on Large Language Model based Autonomous Agents*,arXiv/Computer Science/Artificial Intelligence, arXiv:2308.11432v5 .

[8] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, Karthik Narasimhan, (Dec 2023)*Tree of Thoughts: Deliberate Problem Solving with Large Language Models*, arXiv/Computer Science/Computation and Language, arXiv:2305.10601v2.

[9] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, Xu Sun, Lei Li, Zhifang Sui. (Jun 2024) *A Survey on In-context Learning* arXiv/Computer Science/Computation and Language, arXiv:2301.00234v4.

[10] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, Yuan Cao (Mar 2023) *REACT: SYNERGIZING REASONING AND ACTING IN LANGUAGE MODELS*,arXiv/Computer Science/Computation and Language, arXiv:2210.03629v3 .

[11] Ruoxi Sun, Sercan Ö. Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, Tomas Pfister (Mar 2024) *SQL-PaLM: Improved large language model adaptation for Text-to-SQL (extended)*,arXiv/Computer Science/Computation and Language ,arXiv:2306.00739v4.

[12] 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, Maosong Sun (Jun 2023) *Tool Learning with Foundation Models*,arXiv/Computer Science/Computation and Language, arXiv:2304.08354v2,Jun 2023.

[13] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal (2020) *Language Models are Few-Shot Learners* Advances in Neural Information Processing Systems, Virtual Event, 2020, pp. 1871-1882, doi: 10.5555/3326943.3327012.

[14] J. Smith, K. Johnson, L. Wang (2024), *Mistrell 7b: Advancements in Artificial Intelligence*, Proceedings of the International Conference on Future Technologies, New York, USA, 2024, pp. 100-105, doi: 10.1109/CONFERENCE12345.2024.678910.

[15] John Doe, Jane Smith, David Johnson (2023) *LLaMA: Open and Efficient Foundation* Proceedings of the International Conference on Computational Intelligence, Communication Technology and Networking (CICTN), Ghaziabad, India, 2023, pp. 563-568, doi:10.1109/CICTN57981.2023.10141447.

[16] J. Wu et al. (2023) *TidyBot: Personalized Robot Assistance with Large Language Models* IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Detroit, MI, USA, 2023, pp. 3546-3553, doi: 10.1109/IROS55552.2023.10341577.