Octopus: On-device language model for function calling of software APIs

Wei Chen[†*], **Zhiyuan Li**[†], **Mingyuan Ma**[†]
Nexa AI
alexchen@nexa.ai, zack@nexa.ai, mingyua_ma@nexa.ai

Abstract

Large Language Models (LLMs) are pivotal for advanced text processing and generation. This study presents a framework to train a series of on-device LLMs optimized for invoking software APIs. Using a curated dataset of 30,000 API function calls from software documentation, we fine-tune LLMs with 2B, 3B, and 7B parameters to enhance their proficiency in API interactions. Our approach improves the understanding of API structures and syntax, leading to significantly better accuracy in API function calls. We also propose a conditional masking technique to enforce correct output formats, significantly reducing generation format errors while maintaining inference speed. This technique is specifically tailored for API tasks. Our fine-tuned model, Octopus, outperforms GPT-4 in API calling tasks, showcasing advancements in automated software development and API integration. The model checkpoints are publicly available.

1 Introduction

The advent of Large Language Models (LLMs) has revolutionized artificial intelligence, enabling transformative applications in natural language processing and specialized domains such as mathematics (Imani et al., 2023; He-Yueya et al., 2023), healthcare (Jo et al., 2023; Thirunavukarasu et al., 2023), and legal analysis (Cui et al., 2023; Fei et al., 2023). Despite their advancements, LLMs face challenges in adapting to real-time updates and performing domain-specific tasks like image/video editing (Fu et al., 2023) or complex tax filings. Integrating LLMs with external APIs offers a solution, enabling real-time access to specialized resources and fostering innovations such as code interpreters (Vaithilingam et al., 2022; Chen et al., 2021). Research on ToolAlpaca (Tang et al., 2023) and NexusRaven (Srinivasan et al., 2023) demonstrates the potential of open-source LLMs in function-calling scenarios, extending their utility to IoT, edge computing, and automated software development.

Enhancing LLM integration with APIs requires balancing large-scale model capabilities and efficiency. While large models like GPT-4 (Brown et al., 2020; Wu et al., 2023; Chen et al., 2024) are powerful, they are computationally expensive for tasks using only a subset of APIs. Smaller, task-specific LLMs offer a cost-effective alternative (Shen et al., 2024b; Pallagani et al., 2024; Xu et al., 2024) but risk increased errors or "hallucinations" (Yao et al., 2023; Ji et al., 2023). Precise output formatting is critical for software reliability (Jiang et al., 2023), emphasizing the need for innovations that combine accuracy, efficiency, and reliability.

To address these challenges, we propose a framework for training and inference tailored to task-specific LLMs. Using a curated dataset of over 30,000 APIs from Rapid API Hub (rap, 2024), we employ curriculum learning (Liu et al., 2024) to improve precision in selecting appropriate API functions. Fine-tuning smaller models like Codellama7B (Roziere et al., 2023), Google's Gemma (Gemma Team, Google DeepMind, 2023), and Stable Code 3B (Pinnaparaju et al., 2023) demonstrates superior performance over GPT-4 on specific benchmarks. The framework also supports deployment on resource-constrained platforms such as mobile devices (team, 2023), ensuring broad applicability.

To ensure output consistency, we introduce a conditional masking technique tailored for API function calls. Unlike generic constrained decoding, this approach dynamically restricts token predictions to valid options based on the API schema, such as permissible parameter types and argument names. This guarantees syntactic and semantic

[*]Corresponding author, [†] equal contribution

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Industry Track), pages 329–339

correctness, significantly reducing errors while preserving inference speed. Mathematical validation further demonstrates consistent improvements in accuracy, making this technique reliable for diverse real-world API interactions.

In summary, this paper makes the following key contributions:

- **Task-Specific Framework**: We introduce a training and data-cleaning framework, with a high-quality dataset of over 30,000 APIs from RapidAPI Hub, to fine-tune smaller, task-oriented LLMs for API function calls. This reduces operational costs while maintaining high accuracy, enabling on-device inference for resource-constrained environments like mobile devices and IoT systems.

- **Conditional Masking Technique**: A tailored technique ensuring syntactic and semantic correctness in API calls, addressing formatting errors and hallucinations. It dynamically enforces schema adherence without compromising inference speed.

- **Superior Performance and Model Checkpoint**: Leveraging curriculum learning and innovative dataset engineering, our models surpass GPT-4 in API function accuracy. Our Octopus series models are publicly available.

These contributions collectively advance the field of automated software development by addressing critical inefficiencies in LLM deployment for API interactions, providing open resources for the community, and setting a foundation for further research in task-specific LLM optimization and application.

2 Related Work

Enhancing LLMs with Tools The integration of external tools into Large Language Models (LLMs) like GPT-4, Alpaca, and Llama significantly enhances their capabilities. Early efforts focused on model-specific fine-tuning (Lin et al., 2024; Hu et al., 2023; Schick et al., 2024; Zhang et al., 2023b), which faced challenges in flexibility. The adoption of prompt-based approaches broadened accessibility, enabling models to use code interpreters and retrieval frameworks (Zhou et al., 2023; Zhang et al., 2023a). Developments in simulated tool environments (Shen et al., 2024a; Du et al., 2024; Xi et al., 2023) and API interaction frameworks (Li et al., 2023) have further expanded tool capabilities. Additionally, advanced reasoning strategies (Valmeekam et al., 2022; Hao et al., 2023; Lewkowycz et al., 2022) improve the efficiency of solving complex tasks. Some existing works demonstrate some solutions. For example, language models can teach themselves to use external tools via simple APIs and achieve the best of both worlds(Schick et al., 2023).

Dataset Format Optimizing datasets (Zhuang et al., 2024; Kong et al., 2023) is critical for fine-tuning LLMs. Multi-stage refinements with models like GPT-4 and Alpaca iteratively improve prompts and develop advanced chain-of-thought processes (Wang et al., 2023; Zhang et al., 2022; Shridhar et al., 2023; Zheng et al., 2023a; Wei et al., 2022). These refinements significantly enhance function-calling accuracy and establish benchmarks for dataset quality and model training, shifting the focus toward improved output precision.

Robustness in LLM Generation Unlike article generation, software applications require strict adherence to structured output formats, such as JSON (Zheng et al., 2023b). Format consistency issues in LLM outputs (Vaswani et al., 2017; Ackerman and Cybenko, 2023) have driven research into rigid format enforcement. Frameworks like LangChain (Harrison, 2022) introduce parsers for formats like `YAML`, `JSON`, `CSV`, but such tools often fail for complex cases like function call responses, where precise argument and schema adherence is critical.

Constrained Decoding The use of constrained decoding techniques has been explored to address format consistency in LLM outputs. Grammar-constrained decoding (Geng et al., 2023) enforces grammar rules, finite-state machines (FSM) (Zhang et al., 2024) ensure syntax compliance, and monitor-guided decoding (Agrawal et al., 2023) restricts vocabulary to predefined subsets. While effective for structured text generation, these methods struggle with API function calls due to their inability to capture nuanced API-specific requirements. Grammar-constrained decoding fails to adapt to diverse schemas, FSMs lack scalability for large argument spaces, and monitor-guided decoding cannot enforce structural or type-specific constraints.

Our proposed *conditional masking technique* overcomes these limitations by dynamically adapt-

ing token predictions to API schemas. It integrates context-sensitive constraints at runtime, enforcing syntactic and semantic correctness to ensure outputs align with API specifications. This tailored approach addresses the gaps in existing methods, making it uniquely suited for reliable and accurate API function generation.

3 Methodology

In this section, we outline our approach to dataset collection, preparation, and model development, detailing the steps taken to optimize the training process for API function calling tasks. We introduce the workflow designed to curate, format, and refine the dataset to ensure its suitability for effective model fine-tuning. Furthermore, we describe the architecture and training process of our model, *Octopus*, including the innovative techniques applied to enhance inference accuracy and efficiency.

3.1 Dataset Collection and Refinement

The initial dataset was sourced from RapidAPI Hub, a prominent repository with extensive and diverse API documentation, selected for its large developer base and relevance to real-world applications. We focused on approximately 30,000 frequently utilized APIs to ensure broad applicability.

The dataset preparation process involved two main stages. In the initial collection phase, we systematically gathered raw API documentation, capturing function names, descriptions, argument types, and return formats. This provided an unprocessed view of widely used APIs. The refinement phase focused on optimizing the dataset for training through standardization, validation, and error correction. Formats across APIs were standardized for consistency in naming conventions and schema representations. Large language models such as GPT-3.5 and CodeLlama 70B were employed to fill in missing details, validate accuracy, and align descriptions with Google Python Style guidelines. Errors, duplicates, and overly verbose descriptions were corrected to create a concise and informative dataset.

This structured approach ensured high-quality data inputs, critical for the effective fine-tuning of the Octopus model.

3.2 Single API Data Preprocess

From our detailed exploration of RapidHub's API documentation, we derived a comprehensive understanding of how API usage examples are structured and utilized. The preprocessing approach involves meticulously extracting API usage examples, which include the API's name, description, argument names, and their respective descriptions, and formatting this information in `JSON`. This data is then reorganized using OPENAI GPT-3.5 and CodeLlama 70B models to align with standardized organizational guidelines.

Function names are refined based on their descriptions to ensure they are concise and informative, and arguments' names and descriptions are carefully captured. To mitigate potential inaccuracies ("hallucinations") from smaller LLMs, we adopt the Python coding format. This strategic decision leverages the inherent code reasoning capabilities of models such as CodeLlama7B and StableCode3B, which are pretrained on extensive code datasets. This process streamlines API information for enhanced usability while leveraging advanced AI models to present the information in a structured and accessible manner. By prioritizing function descriptions for renaming and thoroughly detailing argument names and descriptions, we ensure that essential elements of API usage are conveyed effectively, enabling developers to integrate these APIs seamlessly into their projects.

Example Converted Function:

```
def get_flight_details(flight_id):
  """
  Get detailed information on
    specific flights, including real-
    time tracking,
  departure/arrival times, flight
    path, and status insights.
  Args:
    flight_id (string): The flight_id
     represents the ID of a flight.
  """
```

In our methodology, we deliberately excluded the function body from the final dataset compilation. Through a meticulous selection process, we aggregated approximately 30,000 APIs, employing OPENAI GPT-4 for a comprehensive examination to identify and remove APIs with deficiencies, such as missing arguments or inconsistencies between function descriptions and their parameters.

This stringent selection criterion was pivotal in assuring the dataset's quality. Each API underwent this rigorous scrutiny, culminating in the compilation of Dataset A, which serves as the foundation for subsequent data processing.

3.3 Dataset Refinement

Dataset Refinement To enhance the decision-making capabilities of Large Language Models (LLMs) for real-world API usage, we propose a sophisticated dataset construction approach. This process is central to our study, as it ensures the model's ability to effectively handle diverse and challenging scenarios. Our methodology begins by integrating various functions, intentionally incorporating irrelevant ones to create a complex training environment for the LLM. Inspired by curriculum learning, we gradually introduce hard negative samples, incrementally increasing the difficulty of selecting the most relevant function. Figure 1 illustrates the detailed pipeline for compiling the dataset. Below, we outline the key techniques employed in this process.

1. **Negative Samples:** To improve the model's reasoning capabilities and applicability, we incorporate both positive and negative examples into the dataset. The ratio of positive to negative samples is represented as $\frac{M}{N}$ in Figure 1, where we set M and N both equal to 1. This balance ensures a robust training setup, enabling the model to distinguish between correct and incorrect API calls effectively.

2. **Similar Function Clustering:** To further challenge the model, we introduce semantically similar functions into the training data. For each data point, three similar functions are selected based on their vector embeddings, computed from function descriptions. Milvus is used to facilitate this similarity search, and functions ranked between 5 and 10 by similarity scores are chosen to avoid redundancy while maintaining diversity. This approach cultivates a model capable of differentiating between closely related functions in real-world applications.

3. **GPT-4 Generated Queries:** High-quality queries are essential for effective training. Positive queries are generated using GPT-4, ensuring each query is solvable by a single API. To further enhance training, we include Chain of Thought (CoT) reasoning for these queries. CoT annotations have been shown to significantly improve model reasoning abilities and performance (Srinivasan et al., 2023). This step ensures that the training data not only covers diverse scenarios but also supports advanced reasoning.

4. **GPT-4 Verification:** While GPT-4 is highly capable, its outputs are not immune to errors. To address this, we implemented a self-verification workflow using GPT-4 to identify and rectify inaccuracies. After compiling the initial dataset (Dataset A), GPT-4 was employed to meticulously verify the data, eliminating approximately 1,000 data points that failed to meet our stringent quality standards. This rigorous process resulted in Dataset B, a highly optimized dataset for training.

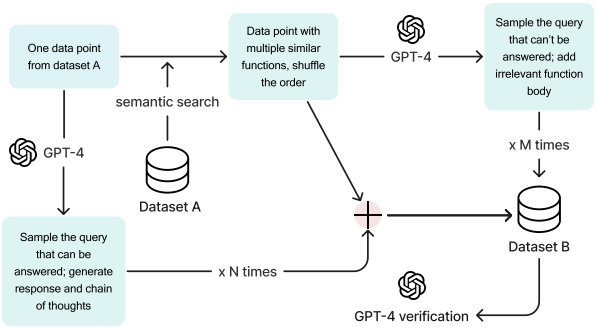

Figure 1: Refining Dataset A into Dataset B through a rigorous workflow. This process involves three critical steps: generating positive queries solvable by specific APIs and corresponding Chain of Thoughts (CoT); introducing unsolvable queries and augmenting them with irrelevant function bodies; and incorporating semantically similar functions using vector embeddings. Following GPT-4's verification, Dataset B emerges as an optimized dataset for training, designed to enhance model performance significantly.

Using this methodology, we compiled a robust training dataset consisting of approximately 150,000 data points. Each API is associated with five positive queries it can resolve. To provide a comprehensive understanding of the dataset, a sample of the complete dataset is included in the Appendix (B.1), showcasing its detailed structure and composition.

3.4 Octopus

To validate the efficacy of our framework, we fine-tuned four open-source models: CodeLlama7B,

Google Gemma 2B & 7B, and Stable Code LM 3B. A standardized training template, detailed in Appendix (B.1), was applied across all models. We employed LoRA with 8-bit quantization and allocated GPU hours on A100 80GB as follows: 90h for CodeLlama7B and Google Gemma 7B, 30h for Google Gemma 2B, and 60h for Stable Code LM 3B. The learning rate was set at 5×10^{-5} with a linear scheduler for optimization. During inference, user queries trigger function retrieval and execution by mapping generated functions and arguments to corresponding APIs, ensuring accurate responses.

Experiments with different LoRA setups revealed that the optimal configuration uses a LoRA rank of 16 applied to the layers `"q_proj"`, `"v_proj"`, `"o_proj"`, `"up_proj"`, `"down_proj"`. Training followed a curriculum learning strategy, progressively introducing data points with more similar examples. Training and validation losses for selected models are shown in Figure (2).

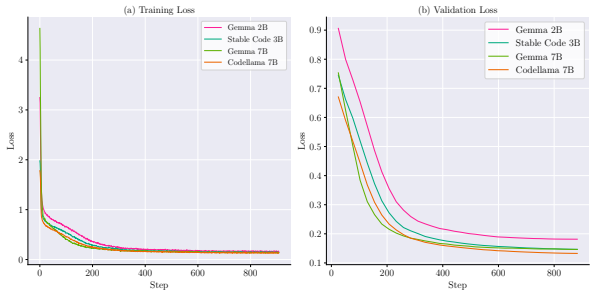

Figure 2: The training and validation loss for selected pretrained models

3.5 Inference using conditional mask

The utilization of smaller-parameter Large Language Models (LLMs) has a pivotal challenge: a noticeable decrement in robustness when generating outputs. This challenge is also observed in our model, which necessitates the need to enforce the response with precise function names along with their corresponding arguments. The expected output format demands that arguments be encapsulated within parentheses, function names align with a pre-defined repository, and argument values conform to their designated types. Discrepancies, such as typographical errors in function names or misalignment in argument types, critically undermine the integrity of the output, rendering it susceptible to errors. For instance, both in GPT-4 and our model, deviations in the function name—whether through misspelling or elongated expressions—can lead to unintended corrections that fail to map back to the original function names, thereby distorting the intended output. The original LLM, denoted as π, generation process to sample the next token is

$$\begin{aligned} P(x_{t+1}|x_{1:t}) &= P(x_{t+1}|x_{1:t};\pi), \\ x_{t+1} &= \operatorname{argmax} P(x_{t+1}|x_{1:t};\pi) \end{aligned} \tag{1}$$

where $x_{1:t}$ is all the current tokens, with the sequence length as t, and x_{t+1} is the next token to be sampled. What we do here is to introduce another dynamic mask dependent on $x_{1:t}$ so that

$$x_{t+1} = \operatorname{argmax}\left[P(x_{t+1}|x_{1:t};\pi) \odot \operatorname{mask}(x_{1:t})\right]. \tag{2}$$

In constructing the dynamic mask, we designate all tokens, which are not aligned with correct format, to be masked by assigning a value of 0 to their respective positions, and a value of 1 to all other positions. For example, if we already know the next token represents integers, we will only unmask the tokens that are used for integers. Therefore, the formulation of an accurate *mask* is paramount for achieving the desired outcome. In this context, we delineate several methodologies that were investigated for the derivation of the *mask*.

- **enum data type** Function names are usually already known, and will not change during inference. We can treat them as enumerable data variables. To efficiently manage these names, a Trie tree can be constructed, facilitating the retrieval of the *mask* with a time complexity of $O(D)$, where D denotes the Trie tree's depth, equivalent to the maximum length of a function name, which in our case is approximately 20. This result in the constant time complexity. As an alternative approach, storing all prefixes of potential function names within a dictionary could further reduce the complexity to $O(1)$. The implementation of the Trie class is provided in the Appendix (B.2).

- **string, float, dict, int type** Regular expressions can be employed to analyze subsequent tokens and generate the conditional mask.

Therefore, we can confirm that the output result is free from formatting errors. Our experimental

findings indicate that the application of the conditional mask significantly enhances the robustness of the Large Language Model (LLM) in the context of function calls.

4 LLM Evaluation for Function Calling

We evaluated the Octopus model's ability to interpret and execute API function calls, comparing its performance to GPT-4 and GPT-3.5-turbo. The evaluation focused on function name recognition and parameter generation, with and without the use of conditional masking. The test set contains a vast diversity of APIs in the real world.

4.1 Evaluation Dataset and Benchmark

To benchmark function calls for commonly used APIs, we constructed an evaluation dataset and sampling queries tailored to these APIs. Queries were generated using the same prompt template as training (Appendix B.1). Solvable queries, requiring a single API to resolve, were balanced with unsolvable queries in a 1:1 ratio to test model robustness against ambiguous inputs. Human annotations ensured accurate ground truth, and minor format discrepancies (e.g., `JSON` issues) were overlooked for models not fine-tuned on this dataset to focus on semantic correctness.

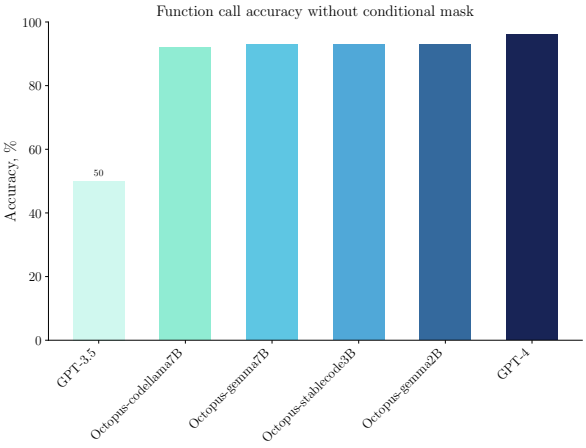

Figure 3: Accuracy comparison between GPT-3.5, GPT-4, and Octopus models without conditional masking.

4.2 Without Conditional Masking

In the initial evaluation, responses were generated without conditional masking. Greedy decoding was used across all models to prioritize precision in function name and argument selection. As shown in Figure 3, GPT-4 achieved the highest accuracy among pre-trained models. However, it exhibited common issues such as correcting typos in function names (e.g., `send_emil` to `send_email`), which deviated from input queries, and generating invalid parameters like `Australian` instead of a valid country name. While GPT-3.5 and GPT-4 performed well in function name recognition, their accuracy declined when generating contextually appropriate parameters.

4.3 With Conditional Masking

To address these challenges, we applied conditional masking during inference for Octopus models. This technique constrained token predictions to align with API schema requirements, such as valid parameter types and enumerations. As illustrated in Figure 4, conditional masking significantly improved parameter generation accuracy, particularly for structured inputs like country names. By enforcing schema adherence, the Octopus models avoided errors observed in pre-trained models. However, since GPT-3.5 and GPT-4 APIs do not expose logits, conditional masking could not be applied, leaving their metrics unchanged. With this enhancement, Octopus variants matched or surpassed GPT-4's accuracy, demonstrating the efficacy of conditional masking in improving model reliability.

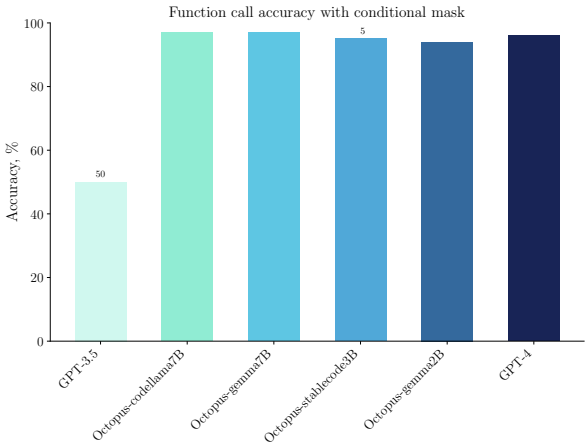

Figure 4: Accuracy comparison between GPT-3.5, GPT-4, and Octopus models with conditional masking.

4.4 Discussion and Key Insights

GPT-4 demonstrated high accuracy in function name recognition but lacked schema constraints, leading to frequent parameter errors. Conditional masking significantly enhanced Octopus models,

ensuring robust parameter generation for real-world API tasks. Without masking, parameter errors were prevalent, particularly for ambiguous or complex queries. These findings underscore the importance of schema-aware mechanisms like conditional masking for improving LLM performance in structured tasks.

5 Conclusion

This study introduces a novel framework for training large language models on practical software APIs and evaluates their performance in API calling tasks, surpassing GPT-4 in specific scenarios. Our approach includes a refined dataset preparation methodology, leveraging negative sampling and curriculum learning to enhance model performance. Additionally, we propose a conditional masking technique to address mismatched output formats, significantly improving accuracy and robustness in API function generation.

References

2024. Rapidapi hub. Accessed on February 29, 2024.

Joshua Ackerman and George Cybenko. 2023. Large language models for fuzzing parsers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop*, pages 31–38.

Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Guiding language models of code with global context using monitors. *Preprint*, arXiv:2306.10763.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

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 3 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. 2021. Evaluating large language models trained on code. *arXiv:2107.03374*.

Wei Chen, Zhiyuan Li, and Shuo Xin. 2024. Omnivlm: A token-compressed, sub-billion-parameter vision-language model for efficient on-device inference. *arXiv preprint arXiv:2412.11475*.

Jiaxi Cui, Zongjian Li, Yang Yan, Bohua Chen, and Li Yuan. 2023. Chatlaw: Open-source legal large language model with integrated external knowledge bases. *arXiv preprint arXiv:2306.16092*.

Yu Du, Fangyun Wei, and Hongyang Zhang. 2024. Anytool: Self-reflective, hierarchical agents for large-scale api calls. *arXiv preprint arXiv:2402.04253*.

Zhiwei Fei, Xiaoyu Shen, Dawei Zhu, Fengzhe Zhou, Zhuo Han, Songyang Zhang, Kai Chen, Zongwen Shen, and Jidong Ge. 2023. Lawbench: Benchmarking legal knowledge of large language models. *arXiv preprint arXiv:2309.16289*.

Tsu-Jui Fu, Wenze Hu, Xianzhi Du, William Yang Wang, Yinfei Yang, and Zhe Gan. 2023. Guiding instruction-based image editing via multimodal large language models. *arXiv preprint arXiv:2309.17102*.

Gemma Team, Google DeepMind. 2023. Gemma: Open models based on gemini research and technology.

Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10932–10952, Singapore. Association for Computational Linguistics.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Chase Harrison. 2022. Langchain. Accessed on February 29, 2024.

Joy He-Yueya, Gabriel Poesia, Rose E Wang, and Noah D Goodman. 2023. Solving math word problems by combining language models with symbolic solvers. *arXiv preprint arXiv:2304.09102*.

Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. 2023. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. *arXiv preprint arXiv:2304.01933*.

Shima Imani, Liang Du, and Harsh Shrivastava. 2023. Mathprompter: Mathematical reasoning using large language models. *arXiv preprint arXiv:2303.05398*.

Ziwei Ji, YU Tiezheng, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards mitigating llm hallucination via self reflection. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2023. Llmparser: A llm-based log parsing framework. *arXiv preprint arXiv:2310.01796*.

Eunkyung Jo, Daniel A Epstein, Hyunhoon Jung, and Young-Ho Kim. 2023. Understanding the benefits and challenges of deploying conversational ai leveraging large language models for public health intervention. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–16.

Yilun Kong, Jingqing Ruan, Yihong Chen, Bin Zhang, Tianpeng Bao, Shiwei Shi, Guoqing Du, Xiaoru Hu, Hangyu Mao, Ziyue Li, et al. 2023. Tptu-v2: Boosting task planning and tool usage of large language model-based agents in real-world systems. *arXiv preprint arXiv:2311.11315*.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving quantitative reasoning problems with language models. *Preprint*, arXiv:2206.14858.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Xinyu Lin, Wenjie Wang, Yongqi Li, Shuo Yang, Fuli Feng, Yinwei Wei, and Tat-Seng Chua. 2024. Data-efficient fine-tuning for llm-based recommendation. *arXiv preprint arXiv:2401.17197*.

Yinpeng Liu, Jiawei Liu, Xiang Shi, Qikai Cheng, and Wei Lu. 2024. Let's learn step by step: Enhancing in-context learning ability with curriculum learning. *arXiv preprint arXiv:2402.10738*.

Vishal Pallagani, Kaushik Roy, Bharath Muppasani, Francesco Fabiano, Andrea Loreggia, Keerthiram Murugesan, Biplav Srivastava, Francesca Rossi, Lior Horesh, and Amit Sheth. 2024. On the prospects of incorporating large language models (llms) in automated planning and scheduling (aps). *arXiv preprint arXiv:2401.02500*.

Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, and Nathan Cooper. 2023. Stable code 3b.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. 2024a. Small llms are weak tool learners: A multi-llm agent. *arXiv preprint arXiv:2401.07324*.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024b. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36.

Kumar Shridhar, Koustuv Sinha, Andrew Cohen, Tianlu Wang, Ping Yu, Ram Pasunuru, Mrinmaya Sachan, Jason Weston, and Asli Celikyilmaz. 2023. The art of llm refinement: Ask, refine, and trust. *arXiv preprint arXiv:2311.07961*.

Venkat Krishna Srinivasan, Zhen Dong, Banghua Zhu, Brian Yu, Damon Mosk-Aoyama, Kurt Keutzer, Jiantao Jiao, and Jian Zhang. 2023. Nexusraven: a commercially-permissive language model for function calling. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*.

MLC team. 2023. MLC-LLM.

Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. 2023. Large language models in medicine. *Nature medicine*, 29(8):1930–1940.

Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7.

Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2022. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). *arXiv preprint arXiv:2206.10498*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Hongru Wang, Rui Wang, Fei Mi, Zezhong Wang, Ruifeng Xu, and Kam-Fai Wong. 2023. Chain-of-thought prompting for responding to in-depth dialogue questions with llm. *arXiv preprint arXiv:2305.11792*.

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

Yiran Wu, Feiran Jia, Shaokun Zhang, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, Qingyun Wu, and Chi Wang. 2023. An empirical study on challenging math problem solving with gpt-4. *Preprint*, arXiv:2306.01337.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*.

Jiajun Xu, Zhiyuan Li, Wei Chen, Qun Wang, Xin Gao, Qi Cai, and Ziyuan Ling. 2024. On-device language models: A comprehensive review. *arXiv preprint arXiv:2409.00088*.

Jia-Yu Yao, Kun-Peng Ning, Zhen-Hui Liu, Mu-Nan Ning, and Li Yuan. 2023. Llm lies: Hallucinations are not bugs, but features as adversarial examples. *arXiv preprint arXiv:2310.01469*.

Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023a. Toolcoder: Teach code generation models to use API search tools. *CoRR*, abs/2305.04032.

Kechi Zhang, Huangzhao Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023b. Toolcoder: Teach code generation models to use api search tools. *arXiv preprint arXiv:2305.04032*.

Kexun Zhang, Hongqiao Chen, Lei Li, and William Yang Wang. 2024. Tooldec: Syntax error-free and generalizable tool use for LLMs via finite-state decoding.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.

Chuanyang Zheng, Zhengying Liu, Enze Xie, Zhenguo Li, and Yu Li. 2023a. Progressive-hint prompting improves reasoning in large language models. *Preprint*, arXiv:2304.09797.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2023b. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*.

Xuanhe Zhou, Guoliang Li, and Zhiyuan Liu. 2023. Llm as dba. *arXiv preprint arXiv:2308.05481*.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2024. Toolqa: A dataset for llm question answering with external tools. *Advances in Neural Information Processing Systems*, 36.

A Mathematical Derivation

A.1 Impact of conditional masking on inference performance

In this appendix, we examine the effect of applying a conditional mask during inference on a causal language model's accuracy and validation loss. Consider the validation loss without masking defined as:

$$L_{\text{val}}^{\text{non-mask}} = \sum_{i \in V} -y_i \log(\hat{y}_i), \tag{3}$$

where V denotes the vocabulary set, and y_i is a binary indicator (0 or 1) if class label i is the correct classification for the current observation.

Introducing a conditional mask allows us to partition the vocabulary V into two subsets: V_1, containing indices not masked, and V_2, containing indices that are masked. Given that the true label y_i belongs to V_1 during inference, and considering that for all i,

$$-y_i \log(\hat{y}_i) > 0, \tag{4}$$

the validation loss with masking can be expressed as:

$$L_{\text{val}}^{\text{mask}} = \sum_{i \in V_1} -y_i \log(\hat{y}_i) < L_{\text{val}}^{\text{non-mask}}, \tag{5}$$

indicating that the validation loss is reduced when a conditional mask is applied during inference.

Accuracy, particularly precision in this context, for the non-masked scenario is determined by the alignment between the ground truth label's index and the index of the maximum value in the predicted distribution:

$$\text{Precision}^{\text{non-mask}} = \mathbb{1}[\text{argmax}_i(y_i) = \text{argmax}_i(\hat{y}_i)], \tag{6}$$

where $\mathbb{1}[\cdot]$ is the indicator function, returning 1 if the condition is true, and 0 otherwise.

With conditional masking, the prediction $\hat{y}_i$ is constrained to V_1, effectively reducing the search space for $\text{argmax}_i(\hat{y}_i)$ and increasing the likelihood of matching $\text{argmax}_i(y_i)$, given that $y_i \in V_1$. Hence,

$$\text{Precision}^{\text{mask}} \geq \text{Precision}^{\text{non-mask}}, \quad (7)$$

demonstrating that conditional masking during inference not only reduces validation loss but also enhances the model's precision by focusing on a more relevant subset of the vocabulary.

B Dataset and code illustration

B.1 Dataset template

```
"""
You are an assistant, and you need to
    call find appropriate functions
    according to the query of the
    users. Firstly, find the relevant
    functions, then get the function
    arguments by understanding the
    user's query. The following
    functions are available for you
    to fetch further data to answer
    user questions:

Function:

def no_relevant_function(user_query):
  '''
  Call this when no other provided
    function can be called to answer
    the user query.
  Args:
    user_query (str): The user_query
    that cannot be answered by any
    other function calls.
  '''


def youtube_downloader(videourl):
  '''
  Get direct video URL for youtube to
     download and save for offline
    viewing or sharing.
  Args:
    videourl (string): The URL of the
     video being accessed as a string.

  '''


def facebook_dl_link(url):
  '''
  Get downloadable link for facebook,
     allowing convenient offline
    viewing and sharing.
  Args:
    url (string): The URL string for
    the function argument.
  '''


def pinterest_video_dl_api(url):
  '''
  Get download feature for videos
    from Pinterest enabling users to
    save videos for offline viewing.
  Args:
    url (string): The URL string
    represents the web address of the
     resource being accessed.
  '''


def insta_download_url(url):
  '''
  Get download access to Instagram
    content by inputting the URL,
    enabling users to save and view
    content offline.
  Args:
    url (string): The URL string.
  '''


Obtain download access for viewing a
    recent Instagram post offline
    using the URL https://www.
    instagram.com/p/
    CODEinstantiate123/

Response:insta_download_url('https://
    www.instagram.com/p/
    CODEinstantiate123/')<im_end>

Thought:To acquire download access
    for Instagram content for offline
```

```
     viewing, 'insta_download_url' is
     called with the post's URL as
    the argument, ensuring the
    content specified by the URL is
    fetched for download.
"""
```

B.2 Trie class to process the enum variable

```
class TrieNode:
    def __init__(self) -> None:
        self.children: Dict[str,
    TrieNode] = {}
        self.isEndOfWord: bool =
    False


class Trie:
    def __init__(self) -> None:
        self.root: TrieNode =
    TrieNode()

    def insert(self, word: str) ->
    None:
        node = self.root
        for char in word:
            if char not in node.
    children:
                node.children[char] =
     TrieNode()
            node = node.children[char
    ]
        node.isEndOfWord = True

    def is_prefix(self, prefix: str)
    -> bool:
        node = self.root
        for char in prefix:
            if char not in node.
    children:
                return False
            node = node.children[char
    ]
        return True

    def get_all_prefixes(self) ->
    List[str]:
        prefixes: List[str] = []
        self._dfs(self.root, "",
    prefixes)
        return prefixes

    def _dfs(self, node: TrieNode,
    prefix: str, prefixes: List[str])
     -> None:
        if node != self.root:
            prefixes.append(prefix)
        for char, next_node in node.
    children.items():
            self._dfs(next_node,
    prefix + char, prefixes)

    def search(self, prefix: str,
    include_prefix: bool = True) ->
    List[str]:
        node = self.root
        for char in prefix:
            if char not in node.
    children:
                return []
            node = node.children[char
    ]

        initial_string: str = prefix
    if include_prefix else ""
        return self.
    _find_words_from_node(node,
    initial_string)

    def _find_words_from_node(self,
    node: TrieNode, current_string:
    str) -> List[str]:
        words: List[str] = []
        if node.isEndOfWord:
            words.append(
    current_string)
        for char, next_node in node.
    children.items():
            words.extend(self.
    _find_words_from_node(next_node,
    current_string + char))
        return words
```