Final Report CSE 291

Ganesh Bannur
gbannur@ucsd.edu

Bhavik Chandna
bchandna@ucsd.edu

Ramsundar Tanikella
rtanikella@ucsd.edu

1 Introduction

The ability of recent large language models (LLMs) to execute commands using natural language has revolutionized how users interact with technology, enabling the development of agentic systems that seamlessly integrate tools and APIs. However, the reliance of these models on cloud-based infrastructure poses critical challenges, including data privacy concerns, reliance on stable internet connectivity, and high latency during real-time operations. These limitations restrict the widespread deployment of LLMs in resource-constrained environments and devices. Our team believes solving this problem is a necessity in today's world where AI technology is not available everywhere. There is a growing significance of integrating AI-powered tools into real-world applications, where scalability, security, and accessibility is ensured.

Motivated by these challenges, our team focused on reproducing [3]. The paper presents a novel framework named TinyAgent to harness the power of task-specific small language models (SLMs) for function calling and orchestration, all while being deployed at the edge. The driving question of this research is

Can a compact language model, trained with targeted datasets, emulate or surpass the function-calling capabilities of much larger models like GPT-4?

This topic is not only academically intriguing but also holds significant real-world implications. If successful, it could revolutionize edge computing by enabling low-latency, secure, and efficient AI interactions on devices ranging from smartphones to IoT devices.

We replicate the experiments on the TinyAgent datasets and find that small LLMs, when finetuned, can match and even exceed the performance of massive LLMs. However finetuning a model on a task-specific dataset also compromises generalizability. To test the effect finetuning has, we also conduct experiments to measure the generalization ability of TinyAgent models. We find that finetuning severely impacts the generalization capability of the models.

Our contributions are

- We replicate the experimental results of [3]
- We conduct experiments to measure the generalization ability of TinyAgent. To do this we paraphrase all the tool names, tool argument names, and tool descriptions in the TinyAgent dataset. We make sure that every paraphrased entity's meaning is equivalent to the original.
- We convert the MacOS application codebase into a research codebase which allows for evaluation. This included changes to make the TinyAgent system run completely on-device without any dependence on cloud APIs.

2 Background

Recently there has been a surge of interest in leveraging LLMs as autonomous agents to perform complex tasks through function calling and tool use [2, 6]. GRANITE-20B-FUNCTIONCALLING [1] is a multi-task learning framework that outperforms other open models on the BFCL v2 benchmark

Preprint. Under review.

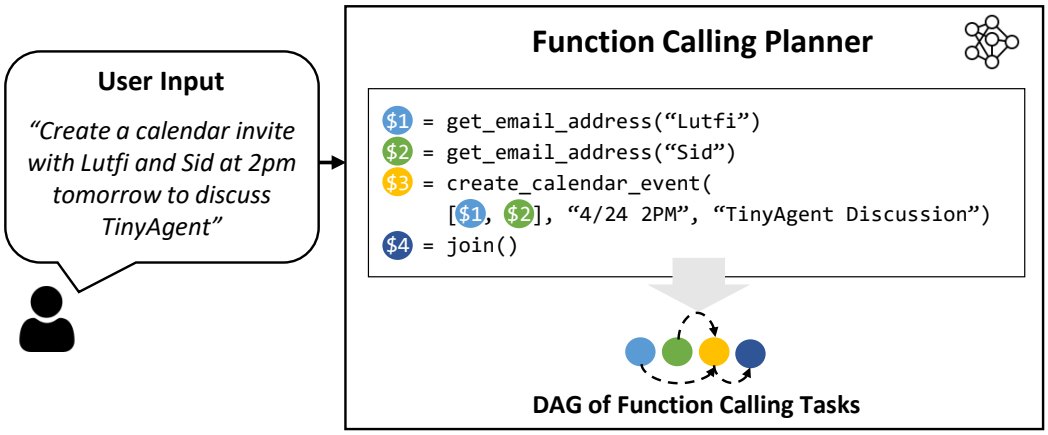

Figure 1: An example of the plan generated by LLMCompiler

Model	Size	Success Rate
GPT-4-Turbo	-	79.08%
WizardLM-2	7 B	41.25%
TinyLlama	1.1 B	12.71%

Table 1: Zero shot performance on the TinyAgent dataset

after being trained on seven fundamental function-calling tasks. Similarly, ToolACE [7] generates diverse tool-learning datasets, allowing the ToolACE-8B model to achieve state-of-the-art results on the BFCL v2 benchmark, rivaling the latest GPT-4 models.

LLMCompiler [5] is a recent framework that enables the LLM to output a function calling plan that includes the set of functions that it needs to call along with the input arguments and their dependencies. Once this function calling plan is generated, it can be parsed into a directed acyclic graph (DAG) and the function calls can be executed in parallel while respecting the dependency order. An example of this is shown in Figure 1. In Figure 1 the LLM first generates a plan consisting of a list of function calls. Dependencies are indicated with placeholders ($1 for example) which indicate which function call's result is required. This list can then be parsed into a DAG with nodes representing function calls and edges indicating the dependencies.

LLMCompiler only considered large LLMs, such as Llama-2-70B, which have complex reasoning abilities. However small LLMs do not have the capacity to perform complex reasoning. The paper we reproduce [3] pushes the boundary by adapting the LLMCompiler framework for small LLMs and enabling efficient inference via parallel function calling.

The motivation for TinyAgent arises from the fact that massive LLMs have the capability to function as agents in new environments without the need for finetuning. In contrast small LLMs lack this ability, highlighting a significant difference in performance. This is further demonstrated in Table 1, which presents the zero shot performance of a few LLMs on the TinyAgent dataset. TinyAgent narrows the performance gap by finetuning small LLMs on an environment-specific dataset. Specifically TinyAgent adapts the LLMCompiler framework for small LLMs.

3 Method

3.1 TinyAgent

TinyAgent is a framework for using small LLMs as agents in specific environments. Table 1 demonstrates that small LLMs cannot perform well in zero shot settings. Hence TinyAgent uses an environment-specific dataset to finetune a small LLM to act as an agent in that environment.

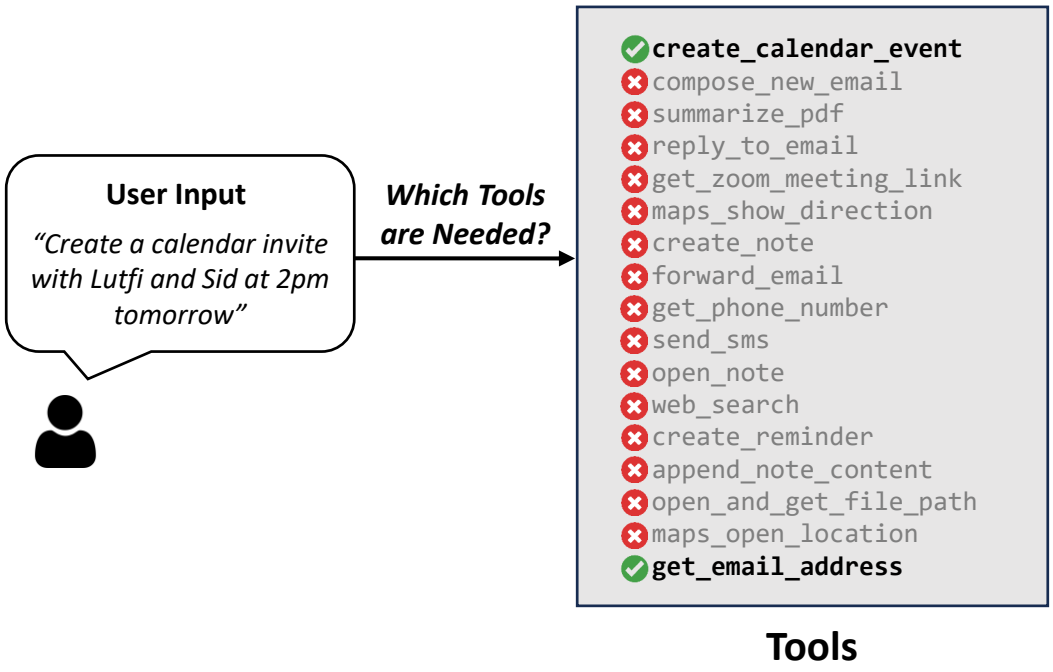

Figure 2: An example where only a small subset of tools are required to fulfill a user query

Finetuning will improve the performance of models on that specific environment but will result in the loss of generalization capability of the models. This tradeoff is worth it if we can get performant agents with far fewer parameters compared to massive LLMs. The paper focuses on the MacOS environment with 16 common tools such as email, SMS, and calendar. Each tool consists of

- Tool name
- Tool arguments: The name and datatype of each argument is specified
- Tool description

To finetune a small LLM for this environment, the paper synthetically generates a dataset using GPT-4-Turbo. The dataset consists of instruction-DAG pairs (80K train, 1K validation, and 1K test). Each pair contains a natural language instruction and the corresponding DAG of function calls and dependencies (as detailed in Section 2). Both the plan in terms of function calls and the DAG obtained after parsing are included. The plan is used as supervision during training and the DAG is used for evaluation. Using this dataset two models are finetuned: TinyLlama-1.1B (instruct-32k) and Wizard-2-7B. The models are finetuned with the tool descriptions of all available tools in the prompt.

3.2 ToolRAG and Quantization

TinyAgent's aim is the on-device deployment of small LLMs for agentic tasks. With this in mind the paper introduces a few optimizations which speed up inference. The most important optimization is ToolRAG. The motivation behind ToolRAG is the fact that not every tool is required for every user query. An example is shown in Figure 2 where only 2 tools are required to fulfill the user query. In general only a small subset of tools will be required to fulfill a user query. So rather than including all the tool descriptions in the LLM's prompt we can select only those tools required to fulfill the user query. The paper first experiments with using basic RAG retrievers to retrieve the relevant tools (treated as documents) given the user query. This does not work well since RAG retrievers only work on semantic similarity (captured by similarity of embedding vectors). However the tools required to fulfill a user query need not be semantically similar to it. For example in Figure 2 the tool `get_email_address` is required even though it has no semantic similarity with the query. This necessitates a model which is trained specifically for selecting the required tools.

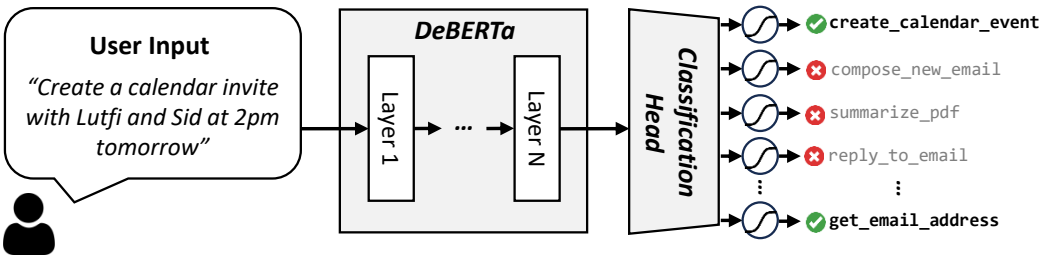

Figure 3: An example of ToolRAG selecting the necessary tools given a user query

The task of selecting tools that are required is easier than agentic behaviour through function calling with planning. Hence an LLM is not required for this task and instead TinyAgent finetunes a small language model to select the necessary tools given the user query. This model is termed ToolRAG given its similarity to how retrieval works in RAG. The paper uses the training data to finetune a DeBERTa-v3-small model [4] to classify whether each tool is necessary for the query. The classification process is shown in Figure 3. This model is used during inference and only the selected tools are included in the LLM's prompt, which reduces prompt size and speeds up inference.

Another optimization introduced by the paper is that they quantize the finetuned TinyAgent LLMs to 4 bits.

3.3 In-Context Examples

One detail which is not mentioned in the paper but implemented in the codebase is the use of in-context examples. Along with the tool descriptions provided in the LLM's prompt, a few in-context examples are also provided. A pool of around 1K examples is given but there is no mention of where they came from. Our best guess is that the validation set of the TinyAgent dataset is used as the pool of examples. Given this pool of examples and the user query, an embedding model is used to embed both the query and examples and a RAG-style retrieval is used to retrieve examples based on embedding similarity. These retrieved examples are appended to the prompt as in-context examples.

3.4 Sensitivity to Tool Names/Descriptions

TinyAgent achieves accurate tool use with small LLMs by finetuning on an environment-specific dataset. As mentioned previously finetuning on an environment-specific dataset is a tradeoff that compromises generalization capability for increased performance. However it isn't clear how severe this tradeoff is i.e. how much generalization capability does a model lose after finetuning. We perform experiments to study the loss in generalization capability. In this experiment we maintain the same set of tools TinyAgent was trained on but paraphrase the tool names, argument names, and descriptions. Crucially we verify that every paraphrased entity has the same meaning as the original. For example a tool named `summarize_pdf` was renamed to `generate_pdf_summary` and its argument `pdf_path` was renamed to `pdf_file_path`. The full list of paraphrased tools can be found at https://github.com/GaneshBannur/TinyAgent/blob/main/src/eval/paraphrased_tools.txt.

If the LLM has actually learnt general tool use then it should have no performance decrease when the tool names, argument names, and descriptions are paraphrased. If the model is able retain its performance this indicates that a model trained for one system (MacOS in the case of the paper) could be adapted to different systems with the same tools but different tool names, arguments names, and descriptions.

4 Experiments

Metrics
In our experiments we measure success rate i.e. the percentage of successful plans. The LLM first generates a plan which consists of an ordered list of tool invocations with corresponding arguments

filled in. Some arguments may be specified as placeholders which are the results of previous tools invocations. This plan is parsed into a DAG (as detailed in Section 2). A plan is considered successful if the predicted DAG and the ground truth DAG are isomorphic.

Experiments Replicating the Paper
We evaluate the success rate of models on the test split of the TinyAgent dataset (see Section 4.2). The full system on which we expect the best performance uses TinyAgent 7B and includes only the relevant tools retrieved by ToolRAG. In this setting the user query is first given to the ToolRAG model which classifies each tool as relevant or irrelevant. Then the names and descriptions of only the relevant tools are included in the prompt of the LLM.

New Experiments
Sensitivity to Tool Names/Descriptions: As detailed in Section 3.4 we study the loss in generalization capability of the finetuned models by paraphrasing tool names, argument names, and descriptions. We run the finetuned 1.1B FP16 model without ToolRAG. We omit ToolRAG since it is trained only on the original tools and is not meant to generalize. When we run without ToolRAG we include all the tool descriptions in the LLM's prompt. Even in this setting results from the paper indicate that performance should not decrease dramatically. So our removal of ToolRAG shouldn't hinder the LLM too much. We run this experiment using the test split of the TinyAgent dataset. Before evaluation we replace every instance of a tool name with its paraphrased version in the ground truth annotations.

Comparison with Other Models: The paper does not compare TinyAgent with other models. However there are other capable small models which are specifically trained for tool use. In the future we would like to compare TinyAgent 1.1B and TinyAgent 7B to models of similar size, `Hammer2.0-1.5B` and `Hammer2.0-7B` from [6]. The comparison will test how well small LLMs trained for general tools use (`Hammer2.0-1.5B`, `Hammer2.0-7B`) can perform without finetuning on the TinyAgent dataset. This experiment will help further the study of the tradeoff between generalization capability and model size. If small LLMs finetuned for general tool use can match TinyAgent then it would indicate that we may not need to sacrifice generalization capability by finetuning on an environment-specific datatset.

Implementation Considerations
The paper missed some important details and since the codebase released was for a MacOS app we had to make many adjustments to convert it into a codebase on which we could perform evaluation.

- **Tokenizers**: TinyAgent models are released without their corresponding tokenizers. For `TinyAgent-1.1B` we use the tokenizer from `TinyLlama-1.1B-32k-Instruct` (https://huggingface.co/Doctor-Shotgun/TinyLlama-1.1B-32k-Instruct). Since the TinyAgent GitHub page compares with this model, we assume this is the base model used for finetuning and use its tokenizer. For the same reason we use the tokenizer from `WizardLM-2-7B` (https://huggingface.co/MaziyarPanahi/WizardLM-2-7B-GGUF) for `TinyAgent-7B`.
- **Multi-Step Planning**: We modify the code to use a single planning step with LLMCompiler. Replanning requires the actual results of running tools on MacOS which, as mentioned previously, makes evaluation challenging and is also unnecessary when evaluating the planner LLM. A single planning step can evaluate the planner LLM without relying on tool execution results.
- **In-Context Examples**: As mentioned in Section 3.3, TinyAgent uses RAG to retrieve in-context examples based on the user query. However it relies on OpenAI's embedding API to generate these embeddings. We replaced the OpenAI embeddings with another model, `nomic-embed-text-v1.5`, to generate the embeddings. `nomic-embed-text-v1.5` was chosen since it has almost the same performance as the OpenAI model `text-embedding-3-small` on MTEB, one of the main embedding benchmarks. We replaced OpenAI embeddings since it would require OpenAI API access and also goes against TinyAgent's aim of fully on-device agents. The embedding model we used can be found at https://huggingface.co/nomic-ai/nomic-embed-text-v1.5.

- **Basic RAG**: There are no details in the paper about which model was used for basic RAG. But similar to the in-context examples, the codebase used OpenAI's `text-embedding-3-small`. We change this to `nomic-embed-text-v1.5` for the same reasons as mentioned for in-context examples. Basic RAG is not actually used in the final TinyAgent system since ToolRAG is used. However we required basic RAG to reproduce ablations.
- **Decoding strategy**: TinyAgent did not specify what decoding strategy was used with their LLMs. We found empirically that using greedy decoding worked best and adpot this for all our experiments.

4.1 Model

We are run experiments with `TinyAgent-1.1B` (https://huggingface.co/squeeze-ai-lab/TinyAgent-1.1B) and `TinyAgent-7B` (https://huggingface.co/squeeze-ai-lab/TinyAgent-7B). Each model has two variants: the original model with 16 bit precision and a quantized model with 4 bit precision. We run experiments with all precisions. When we use ToolRAG we use `TinyAgent-ToolRAG` (https://huggingface.co/squeeze-ai-lab/TinyAgent-ToolRAG).

4.2 Datasets

We use the TinyAgent dataset (https://huggingface.co/datasets/squeeze-ai-lab/TinyAgent-dataset). We run experiments on only the test split of the dataset.

4.3 Baselines

The full TinyAgent system consists of either a 1.1B or 7B LLM in FP16 precision and ToolRAG used to retrieve relevant tools. We compare this to GPT-4-Turbo and also ablations of TinyAgent formed by varying the following aspects

- Weight precision: We compare two weight precisions
  - FP16: Full precision in 16 bits
  - Q4: Quantized to 4 bits
- Tool Retrieval Method: We compare three tools retrieval methods
  - No ToolRAG: All tools are included in the prompt
  - Basic RAG: Using a basic RAG system with `nomic-embed-text-v1.5` as the embedding model
  - ToolRAG: The ToolRAG model is used to retrieve relevant tools

4.4 Code

Our modified codebase is at https://github.com/GaneshBannur/TinyAgent. The codebase released was for a MacOS app and so we encountered many implementation hurdles.

- **Whisper API Integration**: TinyAgent was integrated with Whisper for speech-to-text in the MacOS app. We removed this part of the code since it was not relevant to evaluation and also required OpenAI API access which goes against TinyAgent's aim of fully on-device agents.
- **Mac Deployment Requirement**: The code relied on being deployed on MacOS since it would try to execute MacOS tools. We removed this requirement of MacOS deployment because the evaluation does not need tools to actually run and return a result. Removing this requirement also allowed us to evaluate on any system which had a powerful GPU regardless of the OS.
- **Multi-Step Planning**: As mentioned previously, we modify the code to use a single planning step with LLMCompiler.

Model	Weight Precision	With ToolRAG	Basic RAG	Without ToolRAG	Latency
TinyAgent 1.1B	Q4	63.05% (-17.3)	30.73%	-	2.5s
	FP16	78.79% (-1.27)	75.97% (+1.09)	75.78% (-3.11)	3.7s
TinyAgent 7B	Q4	83.35% (-1.79)	81.05%	80.39%	15.2s
	FP16	84.04% (-0.91)	80.49% (+1.99)	80.28% (-2.81)	16.7s
GPT-4-Turbo*	-	79.08%	-	-	3.9s

Table 2: Comparison of success rate and latency on the TinyAgent test set (Brackets contain deviation from the paper). * indicates results from the TinyAgent paper. - indicates it has not been reported.

- **LLM Output Parsing Robustness**: The parsing of the LLM planner's output had some assumptions which don't always hold true. For example it was assumed that the LLM would always produce a list of actions starting with index 1. When this was not the case the program would crash. We modified the parsing to make it more robust.
- **In-Context Examples**: As mentioned previously, we change the embedding model from OpenAI's `text-embedding-3-small` to the open model `nomic-embed-text-v1.5`

5 Results

5.1 Experiments Replicating the Paper

The results are given in Table 2. On average our evaluation produces results that have a deviation $\leq 3\%$ from the original paper. We observe the same trends as the original results. TinyAgent 7B outperforms TinyAgent 1.1B, which is expected given its larger parameter count. Latency is higher in TinyAgent 7B as compared to TinyAgent 1.1B, which is again expected. In the original paper both TinyAgent 1.1B and 7B outperform GPT-4-Turbo. However in our experiments only TinyAgent 7B outperforms GPT-4-Turbo and TinyAgent 1.1B comes close. The original results in the paper had the Q4 models outperforming the FP16 models. However our results are more of a mixed bag in this regard. The ToolRAG results in the paper indicate that using basic RAG degrades performance compared to no ToolRAG. However we find that basic RAG performs on par with no ToolRAG. Here one of the changes we made comes into play. We use the `nomic-embed-text-v1.5` model to produce embeddings while the original TinyAgent codebase used OpenAI's `text-embedding-3-small`. So perhaps our improved results are due to a `nomic-embed-text-v1.5` being a better embedding model than `text-embedding-3-small`. However the setting using ToolRAG produces the best results which is consistent with the original results.

One major exception to the trends we observe is the TinyAgent 1.1B Q4 model which dropped 17% in success rate from the original paper. In the basic RAG setting as well the 1.1B Q4 model shows a significant performance drop from FP16 (there is no reference number from the original paper for this). We evaluated the 1.1B Q4 model with the same setup as all other models so such a massive drop in performance is unexpected.

5.2 Sensitivity to Tool Names/Descriptions

When we evaluated the TinyAgent 1.1B FP16 model on the paraphrased TinyAgent dataset we saw a severe decrease in performance. The performance decrease was so severe that we weren't able to run evaluations to get success rates. The model hallucinated almost every tool name. Even though the paraphrased names were given in the prompt the model had memorized the original names and continued to use them. This happened even though there were no references to the old tool names in the prompt. Revisiting the example from before, we had a tool originally called `summarize_pdf` which was now called `generate_pdf_summary`. In the outputs we observed the model would always produce `summarize_pdf` in its function calls even though `summarize_pdf` was never mentioned in the prompt. This demonstrates that the model memorizes the tools in the environment-specific dataset it was trained on. Perhaps the toolset being very small (16 tools) made it even easier for models to memorize tools names and ignore what was written in the prompt during training.

6 Conclusion

In this project we adapted TinyAgent's codebase to enable evaluation and local execution, reproduced results from the paper, and performed additional experiments to study TinyAgent's generalization capability. The key takeaways from our project are as follows. It is possible to finetune very small LLMs on a specific-environment and have performance exceeding the zero shot performance of massive LLMs. This is promising and could enable very performant, low latency agents which run on-device and maintain privacy. However finetuning for a specific environment comes at the cost of severely reduced generalization capability. This would mean training a new agent for every new environment. Future work is to evaluate other small LLMs which have been trained for general tool use (such as `Hammer2.0-1.5B`, `Hammer2.0-7B`). This could tell us whether such models can obtain a better tradeoff between performance and generalization capability in small LLMs.

References

[1] Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, GP Bhargav, Maxwell Crouse, Chulaka Gunasekara, et al. Granite-function calling model: Introducing function calling abilities via multi-task learning of granular tasks. *arXiv preprint arXiv:2407.00121*, 2024.

[2] Wei Chen, Zhiyuan Li, and Mingyuan Ma. Octopus: On-device language model for function calling of software apis. *arXiv preprint arXiv:2404.01549*, 2024.

[3] Lutfi Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Tinyagent: Function calling at the edge. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 80–88, 2024.

[4] Pengcheng He, Jianfeng Gao, and Weizhu Chen. Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing. In *The Eleventh International Conference on Learning Representations*, 2023.

[5] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An llm compiler for parallel function calling. *arXiv preprint arXiv:2312.04511*, 2023.

[6] Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, et al. Hammer: Robust function-calling for on-device language models via function masking. *arXiv preprint arXiv:2410.04587*, 2024.

[7] Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, et al. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920*, 2024.