DroidCall: A Dataset for LLM-powered Android Intent Invocation

Anonymous ACL submission

Abstract

The growing capabilities of large language models in natural language understanding significantly strengthen existing agentic systems. To power performant on-device mobile agents for better data privacy, we introduce DroidCall, the first training and testing dataset for accurate Android Intent invocation. With a highly flexible and reusable data generation pipeline, we constructed 10k samples in DroidCall. Given a task instruction in natural language, small language models such as Qwen2.5-3B and Gemma2-2B fine-tuned with DroidCall can approach or even surpass the capabilities of GPT-4o for accurate Android intent invocation. We also provide an end-to-end Android app equipped with these fine-tuned models to demonstrate the Android intent invocation process. The code and dataset are available at https://anonymous.4open.science/r/DroidCall-C100.

1 Introduction

The advent of large language models (LLMs) revolutionizes natural language processing, enabling machines to understand and generate human-like language with unprecedented accuracy. In the realm of mobile computing, this advancement presents a significant opportunity for developing intelligent mobile agents (Li et al., 2024; Zhang et al., 2024b; Wen et al., 2024; Wang et al., 2023a). Specifically, these agents can leverage the rich ecosystem of built-in `intents` (int, 2024) provided by both the operating system and third-party applications on Android devices. These intents serve as a fundamental mechanism for inter-app communication and function invocation, such as sending messages, making phone calls, or triggering specific app features. By harnessing LLMs, mobile agents can interpret diverse and complex user instructions, seamlessly mapping them to the appropriate intents, and therefore automating user interaction with mobile devices.

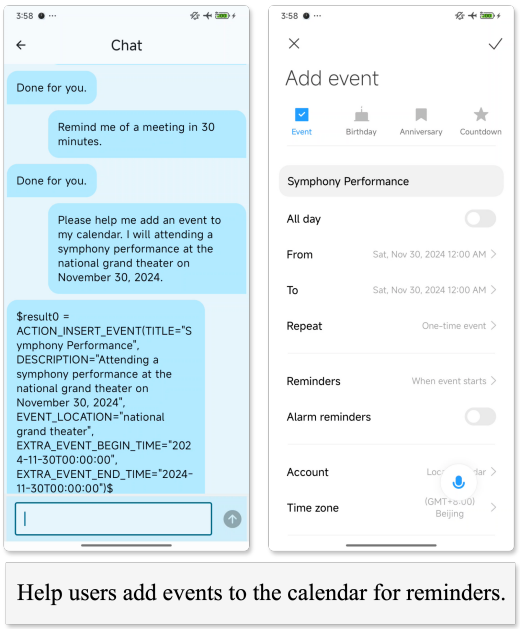


Figure 1: Small language models fine-tuned with DroidCall have the capability to assist users in completing common tasks such as adding events to the calendar.

On-device LLMs are necessary for building mobile agents due to privacy and latency constraints (goo, 2024; Lu et al., 2024b; Yin et al., 2024; Xu et al., 2023b; Yuan et al., 2024). Since user data are processed locally, sensitive information remains on devices, thereby mitigating risks associated with data transmission over networks. Moreover, on-device inference eliminates the need for constant internet connectivity. Various on-device LLM inference optimizations significantly reduce response time (Xu et al., 2024b; Yi et al., 2023a; Xu et al., 2024a), leading to a more responsive and fluid user experience.

However, our investigations reveal a critical challenge: Existing device-affordable LLMs lack the capability of accurate intent invocation. For example, Llama3.2-1B (Dubey et al., 2024) only succeeds in 31.5% and 60.5% of the tasks in zero-shot

and few-shot scenarios, respectively. This limitation is not due to inherent deficiencies in the models themselves but stems from the absence of specialized datasets tailored for this purpose. Existing LLMs are typically trained on broad datasets that do not encompass the specific language patterns and contextual nuances required for accurate intent invocation.

To address this gap, we introduce `DroidCall`, the first open-sourced, high-quality dataset designed for fine-tuning LLMs for accurate intent invocation on Android devices, along with a flexible and reusable data generation pipeline. `DroidCall` comprises an extensive collection of user instructions paired with their corresponding intents, covering a wide array of functionalities across the system and third-party apps while the data generation pipeline automatically generates, validates, and deduplicates data to ensure accuracy and diversity. Unlike existing methods (Wang et al., 2022; Taori et al., 2023; Qin et al., 2023), our approach eliminates the need for manually written seed data, significantly reducing labor.

Evaluation. Based on `DroidCall`, we fine-tuned a series of small language models (SLMs) that are tailored for on-device use. We demonstrate that by fine-tuning models on `DroidCall`, the Android Intent invocation capabilities of these SLMs can be effectively unleashed. Some models can even achieve higher accuracy than GPT-4o using simpler prompts. While prompts for GPT-4o contain an average of 1,367 tokens, models after fine-tuning, achieve this with an average of just 645 tokens. The accuracy of using Gemma2-2B improves from 59% to 85% after fine-tuned on `DroidCall`, while GPT-4o only achieves an accuracy of 77%.

End-to-end demo and open-source. We also provide an end-to-end Android demonstration with the fine-tuned models based on mllm (Yi et al., 2023b), a fast and lightweight multimodal LLM inference engine, which demonstrates the feasibility of our work. The demo is illustrated in Figure 1, which can assist users in completing common operations such as composing emails, setting alarms, making phone calls, and so on. `DroidCall` is available at `https://anonymous.4open.science/r/DroidCall-C100`.

2 Related Work

2.1 LLM-based Agents

LLMs have emerged as a significant advancement in artificial intelligence, particularly in natural language processing. OpenAI's GPT series (Achiam et al., 2023) has led the development of LLMs, which have rapidly gained attention. Open-source LLMs (Yang et al., 2024; Team, 2024; Bai et al., 2023; Dubey et al., 2024; Liu et al., 2024a; Zhu et al., 2024; GLM et al., 2024) have also emerged, with capabilities approaching or rivaling GPT-4. Additionally, models like GPT-4V have extended LLMs with visual capabilities (Yang et al., 2023c; Lu et al., 2024a; Wang et al., 2024c; Liu et al., 2024b), enabling them to handle more complex tasks.

Prompting techniques such as React (Yao et al., 2022), Plan and Solve (Wang et al., 2023b), and ReWOO (Xu et al., 2023a) allow LLMs to plan tasks, use tools, and interact with external environments. These advancements have led to the development of agents like AutoGPT (Yang et al., 2023a), MetaGPT (Hong et al., 2023), and HuggingGPT (Shen et al., 2024b), which can assist humans in various tasks.

2.2 Mobile Device Control Agents

Significant efforts have been made in controlling mobile devices using agents. Early work (Venkatesh et al., 2022; Wang et al., 2023a; Wen et al., 2024) designed UI representations to bridge the gap between GUIs and natural language, enabling models to understand mobile screens. With the advent of multimodal LLMs, agents can now process textual inputs as well as images, audio, or video, allowing them to perceive the environment and accomplish complex tasks. Work such as AppAgent (Yang et al., 2023b) and Mobile Agent (Wang et al., 2024b,a) integrates visual capabilities to implement agents on mobile devices.

However, existing agents have limitations: (1) Most rely on cloud-side LLMs like GPT-4, which raises privacy concerns and fails in poor network conditions. Our work addresses this by deploying SLMs on edge devices. (2) Existing agents simulate human actions (e.g., tap and swipe) to operate devices, which is inefficient and error-prone. We propose intent invocation through function calling as a more efficient and accurate approach. For example, instead of navigating the UI to set an alarm, the agent directly communicates the intent

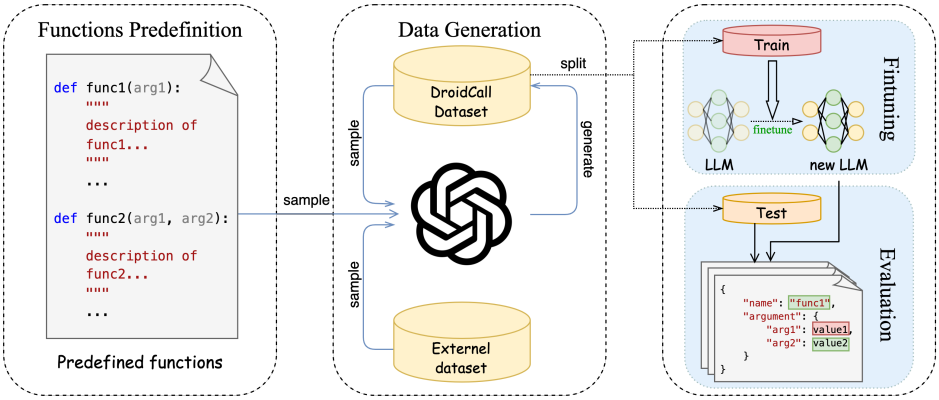


Figure 2: Workflow of DroidCall, which consist of three key phases:(1) Functions Predefinition; (2) Data Generation; (3) Finetuning and Evaluation.

to the app. Our work abstracts intent invocation as function calling, enabling operations on Android without UI interactions.

2.3 LLMs for Function Calling

LLMs possess reasoning capabilities that enable function calling when needed. Toolformer (Schick et al., 2024) pioneered this area by teaching LLMs to use tools during interactions, demonstrating the feasibility of function calling and providing a framework for subsequent research. To equip models with function-calling capabilities, substantial data is often required. Self-Instruct (Wang et al., 2022) shows that LLMs like GPT can generate large volumes of fine-tuning data. Following this paradigm, efforts like (Qin et al., 2023; Tang et al., 2023; Patil et al., 2023; Kim et al., 2023) have generated extensive function-calling data for fine-tuning. Additionally, works such as APIGen (Liu et al., 2024d), ShortcutsBench (Shen et al., 2024a), and ToolACE (Liu et al., 2024c) focus on dataset construction. AgentOhana (Zhang et al., 2024a) standardizes data formats and designs training pipelines for effective agent learning.

In our work, we construct a reusable and customizable data generation pipeline, focusing on Android intent invocation to achieve better edge performance than GPT-4o. We also provide simple methods for fine-tuning and evaluation. Similar works like TinyAgent (Erdogan et al., 2024) and Octopus (Chen and Li, 2024) implement function-calling agents on mobile devices, but TinyAgent is specifically designed to target operations on macOS systems, meaning it can only function on Apple computers, and Octopus requires model architecture adjustments. Neither provides code for data generation or fine-tuning.

3 DroidCall Dataset and Workflow

In this section, we introduce the overall workflow of DroidCall, which comprises three key phases as shown in Figure 2: Function Predefinition, Data generation, Finetuning and Evaluation. In §3.1, we first introduce Android intent, a key mechanism of Android. Based on the common intents in Android, we manually predefine 24 functions that can assist users in performing some common operations on Android. In §3.2, we detail our method for generating the DroidCall dataset, the first open-sourced dataset for Android intent invocation. Our method requires minimal human supervision and can be easily extended. In §3.3, we describe how we fine-tune LLMs and evaluate their performance. §3.4 shows an end-to-end demonstration of device control using fine-tuned LLMs with DroidCall.

3.1 Collecting Android Intents

In Android development, an intent is a messaging object used to request actions from app components. It facilitates communication between components like activities, services, and broadcast receivers. Android intents are categorized into two types:

- **Explicit Intents** specify the exact component to start, typically used for internal app communication.
- **Implicit Intents** declare a general action, allowing any compatible component to respond, making them ideal for inter-app interactions.

The goal of DroidCall is to enable models to perform function calling on Android devices for

common operations. Implicit intents are particularly suitable for this purpose, as they effectively express user intentions and utilize system resources efficiently. To construct the DroidCall dataset, we review the Android official documentation (com, 2024) and select frequently-used intents, encapsulating them into functions that cover common operations, including but not limited to alarm configuration, email composition, and web searching.

3.2 Dataset Generation

In this section, we present a detailed description of the DroidCall dataset generation process. We first introduce the key components utilized in data generation: the *sampler*, *collector*, *LLM* and *filter* components. Subsequently, we elaborate on the critical phases of data generation: function predefinition, seed data generation, and data generation. The entire dataset generation process leverages GPT-4-turbo as the underlying language model. Figure 3 shows an overall workflow of data generation.

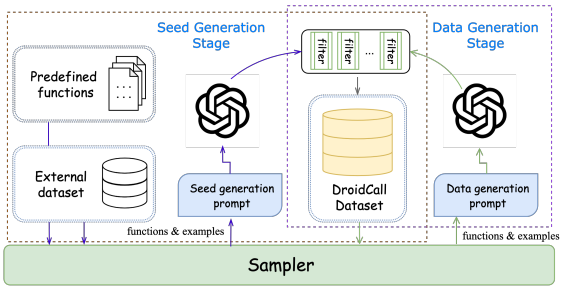


Figure 3: Details of data generation in DroidCall. To avoid manually creating seed data, DroidCall initially samples examples from an external dataset to generate its first set of data. Subsequently, the data is used as seed data to continuously generate new data, thereby eliminating the need for laborious manual work. All the generated data will go through a set of customized filters to ensure the correctness of data formats and the diversity of the data.

3.2.1 Key Components of Generation Pipeline

The data generation pipeline consists of four key components: *Sampler*, *LLM*, *Filter*, and *Collector*.

Sampler. The sampler takes multiple data sources (e.g., lists, jsonl files) as input, samples data according to a specific strategy, and organizes it into a user-defined format for output.

LLM. The LLM is the core engine for data generation. Using the self-instruct paradigm (Wang et al., 2022), we integrate sampled data into prompt templates and generate data via the LLM. In this work, GPT-4-turbo is used as the LLM.

Filter. Filters process the LLM's output, extracting structured data, discarding improperly formatted data, and removing highly similar data. The framework supports custom filters for flexible data processing.

Collector. The collector coordinates the pipeline. It retrieves data from the sampler, integrates it into prompt templates, generates raw data via the LLM, processes the data through filters, and collects the final results.

3.2.2 Functions Predefinition

Automated extraction of intents from the Android Open Source Project (AOS, 2024) is complex due to the dynamic nature of the Android platform. To avoid these challenges, we predefine 24 functions covering common Android operations, utilizing common intents for their implementation. These functions act as an interface between the LLM and the intents, hiding intent details from the LLM. This approach ensures compatibility across different Android versions, as the LLM only needs to learn the functions, while the underlying intent implementations can be adapted as needed. The predefined functions support operations such as:

- **Scheduling Assistant**: Set alarms/timers, insert calendar events.
- **Contact Management**: Add contacts, make phone calls.
- **Common Operations**: Internet search, map search, open camera, adjust settings.
- **Messaging Services**: Compose text messages or emails.

In our framework, functions are predefined similarly to ordinary Python functions. We write function signatures and provide Google-style docstrings (Goo, 2024), from which structured information is automatically extracted. The extracted data format is shown in Listing 1.

```
{
  "name": "func1",
  "description": "This function is ...",
  "arguments": {
    "arg1": {
        "description": "This arg is...",
        "type": "<type>",
        "required": "true or false",
        "default": "<default_value>"
    },
    "arg2": ...
  },
  "returns": {
    "type": "...",
    "description": "..."
  },
```

```
  "example": [...]
}
```

Listing 1: Extracted function. "returns" field and "example" field are optional.

3.2.3 Data generation

We follow the self-instruct paradigm (Wang et al., 2022; Taori et al., 2023) to build our data generation pipeline, which consists of two stages: seed generation and data generation.

Seed Generation Stage. High-quality seed data is crucial for guiding LLMs in synthetic data generation. To avoid manual effort, we automatically generate seed data by leveraging existing function-calling datasets. Specifically, we sample data from xlam-function-calling-60k (Liu et al., 2024d) and prompt the LLM to generate user queries and calling examples for our predefined functions. These seeds are used in the subsequent data generation stage.

Data Generation Stage. In this stage, we use the self-instruct paradigm to generate more data. For each predefined function, we extract examples from the seed data and prompt the LLM to produce additional user queries and function-calling examples. The generated data follows the format shown in Listing 2

```
{
  "query": "user query here",
  "answers": [
    {
        "id": id,
        "name": "func_name",
        "arguments": {
            "arg1": "value1",
            ...
        }
    }, ...
  ]
}
```

Listing 2: An example of generated data

To ensure data quality, we apply three filters sequentially:

JsonExtractor: Extracts JSON data from LLM output using a syntax parser.

FormatFilter: Ensures the extracted JSON matches the required format.

SimilarityFilter: Filters out highly similar queries using the LCS ROUGE score (Lin, 2004), discarding data with an F-measure value above 75%.

We generate two types of function-calling data:

- **Simple**: User queries requiring a single function call. Listing 3 shows an example:

```
{
  "query": "Wake me up at 8:30",
  "answers": [
    {
      "id": 0,
      "name": "ACTION_SET_ALARM",
      "arguments": {
        "EXTRA_HOUR": 8,
        "EXTRA_MINUTE": 30
      }
    }
  ]
}
```

Listing 3: Simple call example

- **Complex**: User queries requiring multiple function calls. Listing 4 shows an example:

```
{
  "query": "Set a timer for 30 minutes and
       dial 123456",
  "answers": [
    {
      "id": 0,
      "name": "ACTION_SET_TIMER",
      "arguments": {
        "duration": "30 minutes"
      }
    },
    {
      "id": 1,
      "name": "dial",
      "arguments": {
        "phone_number": "123456"
      }
    }
  ]
}
```

Listing 4: Complex call example

The `DroidCall` dataset consists of 10,000 training and 200 test entries. All prompt templates are provided in Appendix A.

3.3 Fine-tuning SLMs with `DroidCall`

Models. We fine-tuned a series of SLMs using the `DroidCall` dataset, including PhoneLM-1.5B (Yi et al., 2024), Qwen2.5-1.5B, Qwen2.5-3B (Yang et al., 2024; Team, 2024), Llama3.2-1B, Llama3.2-3B (Dubey et al., 2024), MiniCPM3-4B (Hu et al., 2024), Phi3.5-3.8B (Abdin et al., 2024) and Gemma2-2B (Team et al., 2024).

Modeling function-calling tasks. We treat function calling as an instruction-following task, where the model's input includes the *user query*, *available function descriptions*, and *task instructions*, and the output is a *specific representation for calling a function*.

To avoid performance degradation caused by mismatched formats, we reuse the model's chat template instead of designing a unified input-output format. Most models are fine-tuned for chat tasks involving three roles: system, user, and assistant. We place the *user query* and *available function*

descriptions in the system and user prompts, and the function-calling result in the assistant output. This approach aligns the fine-tuning data with the model's existing knowledge, ensuring better performance.

Setups. We formatted the `DroidCall` dataset into the chat format, resulting in 10K training samples. We fine-tuned the model using LoRA (Hu et al., 2022) with a rank of 8, alpha of 16, and a linear learning rate scheduler (learning rate: 1.41e-5, warmup ratio: 0.1). Training ran for 24 epochs, with the best checkpoint selected. Prompt format details are provided in Appendix B.

3.4 Putting It All Together

Using the `DroidCall` dataset, we equip SLMs with Android intent invocation capabilities. To verify its effectiveness, we developed an Android application, whose design is shown in Figure 4. The demo consists of two key components:

Retriever: Retrieves the most relevant functions using GTE (Li et al., 2023) for word embeddings and ObjectBox (obj, 2024) as the vector database. When a user query arrives, GTE generates embeddings, and ObjectBox retrieves the relevant functions.

Intent Invocation Model: Takes the user query and retrieved functions as input, and outputs the function calls to fulfill the query. We use PhoneLM-1.5B (Yi et al., 2024) fine-tuned on the `DroidCall` dataset for this purpose.

All model inference is performed on mobile devices using mllm (Yi et al., 2023b), a fast and lightweight multimodal LLM inference engine for edge devices. Figure 1 illustrates an example of the end-to-end demo, where the fine-tuned model assists users in adding an event to the calendar.

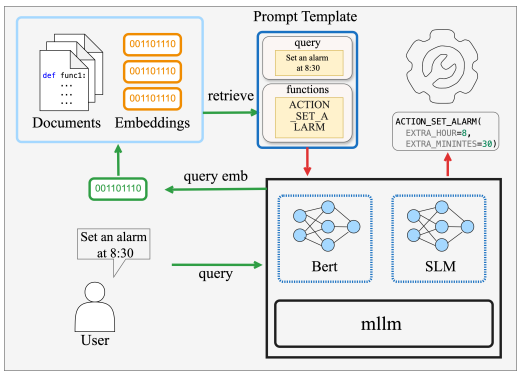


Figure 4: Design of our demo.

4 Experiments

We first explored the impact of prompt designs on model fine-tuning, selecting an optimal format based on prompt length and model performance. We then demonstrated that the `DroidCall` dataset outperforms general function-calling datasets for Android intent invocation. Finally, we present results showcasing the effectiveness of models fine-tuned with `DroidCall`.

Metrics. We introduce two metrics to evaluate function-calling performance: *Accuracy* and *Soft Accuracy*.

- *Accuracy.* Measures the model's ability to perfectly match ground-truth function calls. A sample is correct only if the model's output matches the ground truth in both function identity and parameter values:

$$Acc = \frac{N_{\text{perfect}}}{N_{\text{total}}}$$

- *Soft Accuracy.* Evaluates partially correct function calls by calculating the proportion of accurately predicted parameters. The metric is averaged across all function calls:

$$Acc_{\text{soft}} = \frac{1}{F}\sum_{i=1}^{F}\frac{P_{\text{correct},i}}{P_{\text{total},i}}$$

For parameters like *title* or *subject*, semantic consistency is sufficient. We use RoBERTa (Liu et al., 2019) to measure semantic similarity, with a threshold of 0.75 for correctness.

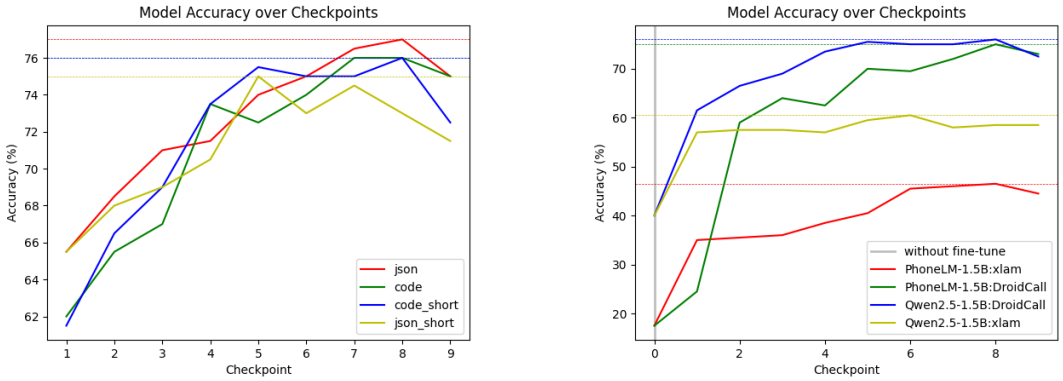
We evaluate SLMs using the 200 test entries from `DroidCall`. During testing, we use a fake retriever that always retrieves ground-truth functions to isolate the impact of the retriever.

4.1 Effect of Different Prompts

Prompt	Average Number of Tokens
code_short	645.195
json_short	950.340
code	931.555
json	1367.905

Table 1: Average number of tokens of different prompts.

In § 3.3, we described the model input components: *user query*, *available function descriptions*, and *task instructions*, with the output being a *specific representation for calling a function*. While



(a) Accuracy of *Qwen2.5-1.5B-Instruct* on different prompts

(b) Different models fine-tuned on different datasets

Figure 5: Figure 5(a) illustrates the performance of *Qwen2.5-1.5B-Instruct* after fine-tuning under different prompt formats. Figure 5(b) shows the performance of *PhoneLM-1.5B* and *Qwen2.5-1.5B-Instruct* after finetuning on different datasets.

the *user query* is user-provided, we designed the remaining components to evaluate their impact on fine-tuning performance.

json. A minimalist design using JSON for *available function descriptions* and function call representations. JSON was chosen for its simplicity.

code. Leveraging the prevalence of code data in pre-training, we used *docstrings* for function descriptions and *Python function calls* for function representations. This aligns with pre-training data, potentially improving model comprehension.

short. We hypothesized that fine-tuned models might not require explicit *task instructions*. Thus, we removed *task instructions* from the *json* and *code* formats, resulting in *json_short* and *code_short*.

We fine-tuned the Qwen2.5-1.5B-Instruct model using four prompt formats. Figure 5(a) shows the accuracy across nine checkpoints. While the *json* format performed slightly better, the *code_short* format achieved comparable results with significantly fewer tokens, as shown in Table 1. Based on these findings, we selected the *code_short* format for subsequent fine-tuning experiments.

4.2 Effectiveness of `DroidCall`

To verify that the `DroidCall` dataset can achieve better results in the task of controlling Android phones through Android Intent invocation, we compared the performance of the Qwen2.5-1.5B-Instruct and PhoneLM-1.5B models after fine-tuning on the `DroidCall` dataset and xlam-function-calling-60k (Liu et al., 2024d), a general function calling dataset.

To eliminate the influence of prompt design, we formatted both the xlam-function-calling-60k and `DroidCall` datasets using the *code_short* format. The xlam-function-calling-60k dataset comprises 60k data points, while `DroidCall` contains 10k. To ensure an equivalent number of training data instances, we trained the model for 4 epochs on the xlam-function-calling-60k dataset and for 24 epochs on `DroidCall`. We selected 9 checkpoints for testing, and the results are presented in Figure 5(b). Note that the 0th checkpoint in the figure represents the model's performance when directly evaluated with the *code* format of prompts before any fine-tuning took place.

From the experimental results, we can observe that, regardless of the dataset used, accuracy improves with the fine-tuning process. However, the model fine-tuned with the xlam-function-calling-60k dataset quickly reaches a plateau. In contrast, the improvement brought by using the `DroidCall` dataset is significantly more substantial.

It is evident that when a model is required to perform a specific task, a dataset constructed for that task, such as `DroidCall`, can yield better results compared to a general-purpose dataset. Furthermore, from Figure 5(b), we can discern that initially, Qwen's capabilities are significantly higher than PhoneLM's. However, by the end of the fine-tuning process, PhoneLM's performance is on par with Qwen's. We speculate that initially, PhoneLM's Supervised Fine-Tuning (SFT) and alignment were not as effective as Qwen's, pre-

Model	Size	Zero-Shot		Few-Shot		Fine-Tuning	
		Acc	Acc_{soft}	Acc	Acc_{soft}	Acc	Acc_{soft}
PhoneLM-1.5B	1.5B	17.5	17.5	55.5	62.8	75	86.1
Qwen2.5-1.5B-Instruct	1.5B	61	76.6	64.5	81	76	90.3
Qwen2.5-3B-Instruct	3B	62	79.4	71	86.1	83	93.5
Qwen2.5-Coder-1.5B	1.5B	42.5	48.8	65.5	81.6	82	93.2
Gemma2-2B-it	2B	59	77.2	67.5	83.7	**85**	**93.9**
Phi-3.5-mini-instruct	3.8B	62	77.8	67.5	82.1	83.5	93.8
MiniCPM3-4B	4B	67	84.3	75	89.6	74.5	82.3
Llama3.2-1B-Instruct	1B	31.5	37.7	60.5	76.3	75.5	87.3
Llama3.2-3B-Instruct	3B	66.5	79.8	72	87.2	82	92.7
GPT-4o (2024-08-06)		77	89.1	80.5	91.5		
GPT-4o-mini (2024-07-18)		71.5	86.6	76	88.6		

Table 2: Evaluation of different models. Our fine-tuned model achieves superior performance compared to GPT-4o, utilizing only half the prompt length and a compact 2 billion parameters.

venting it from leveraging its pre-trained knowledge efficiently. The `DroidCall` dataset, however, aids the model in learning to utilize its pre-trained knowledge to control Android devices. Since the pre-trained knowledge of both PhoneLM and Qwen is comparable, they eventually reach a similar level of performance.

4.3 Performance of Different SLMs

To test the Android intent invocation capabilities of some existing SLMs tailored for the edge scenario and further verify the effectiveness of `DroidCall`, we tested the Acc and Acc_{soft} of a few models under three conditions: zero-shot, few-shot, and after fine-tuning. When testing the models' zero-shot and few-shot performance, we used *json* format prompts for both, as the *json-short* and *code-short* formats lack *task instructions*, which prevents the model from finishing the task. In comparison, the *json* format has been found to be more effective than the *code* format.

The experimental outcomes, as depicted in Table 2, provide a comprehensive overview of the Android intent invocation capabilities across various models. Judging from the zero-shot results, there is a significant performance variation among different models. The zero-shot scenario is a critical test of a model's ability to complete tasks based on instructions without having seen relevant examples. We believe the primary reason for the differences in zero-shot performance among models lies in the effectiveness of their SFT and alignment. These training stages determine whether the model can develop strong instruction-following capabilities. It is also observable that all models exhibit improved performance under few shots. We credit the performance boost to the models' improved use of their knowledge from pretraining.

After fine-tuning with `DroidCall` using *code_short* format, there is a significant improvement in the model's performance. Additionally, during inference, the model only requires a prompt that essentially consists of the *user query* and *available function descriptions*, which greatly reduces the prompt length compared to the zero-shot and few-shot scenarios.

5 Conclusion

In this paper, we introduce `DroidCall`, a novel dataset specifically engineered to enhance the Android intent invocation capabilities of LLMs. Our approach diverged from conventional cloud-based models, focusing instead on on-device deployment to address privacy concerns inherent in mobile environments. In our work, we (1) build a highly customizable and reusable data generation pipeline, (2) construct `DroidCall`, a first-of-its-kind open-sourced dataset for Android intent invocation based on the pipeline, (3) fine-tune a series of models tailored for edge devices, enabling them to approach or even surpass the performance of GPT-4o in the specific task of intent invocation and (4) implement an end-to-end demo with mllm. Our work demonstrates the potential applications of small models on the edge. We have open-sourced all the code of the data generation, fine-tuning, and evaluation.

Limitations

While our approach demonstrates promising results, it has several limitations that warrant further investigation.

Data Quality and Generalizability. Our method relies heavily on the quality of the generated data, which may introduce biases or inaccuracies. For instance, the synthetic data generated by LLMs may not fully capture the diversity of real-world scenarios, potentially limiting the model's generalization ability. In this work, we prioritize generating high-quality, task-specific data for Android intent invocation, which allows us to mitigate some of these issues within the narrow scope of our target domain. However, broader generalization to more diverse or complex scenarios remains a challenge. Future work could explore hybrid data generation techniques, combining synthetic data with real-world user interactions, to improve both diversity and accuracy.

Scalability of the Method. Our approach requires predefined functions, which limits its adaptability to new tasks without significant manual effort. This limitation is partially offset by the modular design of our pipeline, which allows for easy extension to new functions within the Android ecosystem. In the context of this work, we focus on a curated set of common Android intents, where predefined functions are sufficient to cover most use cases. However, for more dynamic or open-ended tasks, this approach may not scale effectively. Future research could investigate methods for automatically discovering and defining new functions, potentially leveraging unsupervised or semi-supervised learning techniques to reduce manual intervention.

References

2024. Aosp. https://source.android.com/.

2024. Common intents. https://developer.android.com/guide/components/intents-common.

2024. Google ai edge sdk for gemini nano. https://developer.android.com/ai/aicore.

2024. Google-style docstrings. https://google.github.io/styleguide/pyguide.html#381-docstrings.

2024. intent. https://developer.android.com/reference/android/content/Intent.

2024. Objectbox: Fast and efficient database with vector search. https://github.com/objectbox/objectbox-java.

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Wei Chen and Zhiyuan Li. 2024. Octopus v2: On-device language model for super agent. *arXiv preprint arXiv:2404.01744*.

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

Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2024. Tinyagent: Function calling at the edge. *arXiv preprint arXiv:2409.00608*.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. 2024. Minicpm: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*.

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

Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, et al. 2024. Personal llm agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459*.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024a. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2024b. Visual instruction tuning. *Advances in neural information processing systems*, 36.

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

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *Preprint*, arXiv:1907.11692.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, et al. 2024d. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *arXiv preprint arXiv:2406.18518*.

Haoyu Lu, Wen Liu, Bo Zhang, Bingxuan Wang, Kai Dong, Bo Liu, Jingxiang Sun, Tongzheng Ren, Zhuoshu Li, Hao Yang, et al. 2024a. Deepseek-vl: towards real-world vision-language understanding. *arXiv preprint arXiv:2403.05525*.

Zhenyan Lu, Xiang Li, Dongqi Cai, Rongjie Yi, Fangming Liu, Xiwen Zhang, Nicholas D Lane, and Mengwei Xu. 2024b. Small language models: Survey, measurements, and insights. *arXiv preprint arXiv:2409.15790*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

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.

Haiyang Shen, Yue Li, Desong Meng, Dongqi Cai, Sheng Qi, Li Zhang, Mengwei Xu, and Yun Ma. 2024a. Shortcutsbench: A large-scale real-world benchmark for api-based agents. *arXiv preprint arXiv:2407.00132*.

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.

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

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving open language models at a practical size. *arXiv e-prints*, pages arXiv–2408.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Sagar Gubbi Venkatesh, Partha Talukdar, and Srini Narayanan. 2022. Ugif: Ui grounded instruction following. *arXiv preprint arXiv:2211.07615*.

Bryan Wang, Gang Li, and Yang Li. 2023a. Enabling conversational interaction with mobile ui using large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–17.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024a. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *arXiv preprint arXiv:2406.01014*.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024b. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023b. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*.

Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, et al. 2024c. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. *arXiv preprint arXiv:2409.12191*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pages 543–557.

Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. 2023a. Rewoo: Decoupling reasoning from observations for efficient augmented language models. *arXiv preprint arXiv:2305.18323*.

Daliang Xu, Wangsong Yin, Xin Jin, Ying Zhang, Shiyun Wei, Mengwei Xu, and Xuanzhe Liu. 2023b. Llmcad: Fast and scalable on-device large language model inference. *arXiv preprint arXiv:2309.04255*.

Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2024a. Empowering 1000 tokens/second on-device llm prefilling with mllm-npu. *arXiv preprint arXiv:2407.05858*.

Mengwei Xu, Wangsong Yin, Dongqi Cai, Rongjie Yi, Daliang Xu, Qipeng Wang, Bingyang Wu, Yihao Zhao, Chen Yang, Shihe Wang, et al. 2024b. A survey of resource-efficient llm and multimodal foundation models. *arXiv preprint arXiv:2401.08092*.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Hui Yang, Sifu Yue, and Yunzhong He. 2023a. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224*.

Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023b. Appagent: Multimodal agents as smartphone users. *arXiv preprint arXiv:2312.13771*.

Zhengyuan Yang, Linjie Li, Kevin Lin, Jianfeng Wang, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. 2023c. The dawn of lmms: Preliminary explorations with gpt-4v (ision). *arXiv preprint arXiv:2309.17421*, 9(1):1.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. 2023a. Edgemoe: Fast on-device inference of moe-based large language models. *arXiv preprint arXiv:2308.14352*.

Rongjie Yi, Xiang Li, Qichen Qiu, Zhenyan Lu, Hao Zhang, Daliang Xu, Liming Yang, Weikai Xie, Chenghua Wang, and Mengwei Xu. 2023b. mllm: fast and lightweight multimodal llm inference engine for mobile and edge devices.

Rongjie Yi, Xiang Li, Weikai Xie, Zhenyan Lu, Chenghua Wang, Ao Zhou, Shangguang Wang, Xiwen Zhang, and Mengwei Xu. 2024. Phonelm:an efficient and capable small language model family through principled pre-training. *Preprint*, arXiv:2411.05046.

Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. 2024. Llm as a system service on mobile devices. *arXiv preprint arXiv:2403.11805*.

Jinliang Yuan, Chen Yang, Dongqi Cai, Shihe Wang, Xin Yuan, Zeling Zhang, Xiang Li, Dingge Zhang, Hanzi Mei, Xianqing Jia, et al. 2024. Mobile foundation model as firmware. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pages 279–295.

Jianguo Zhang, Tian Lan, Rithesh Murthy, Zhiwei Liu, Weiran Yao, Juntao Tan, Thai Hoang, Liangwei Yang, Yihao Feng, Zuxin Liu, et al. 2024a. Agentohana: Design unified data and training pipeline for effective agent learning. *arXiv preprint arXiv:2402.15506*.

Li Zhang, Shihe Wang, Xianqing Jia, Zhihan Zheng, Yunhe Yan, Longxi Gao, Yuanchun Li, and Mengwei Xu. 2024b. Llamatouch: A faithful and scalable testbed for mobile ui automation task evaluation. *arXiv preprint arXiv:2404.16054*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A Data Generation Prompts

At the beginning of data generation, we first generate seed data. The prompt used to generate seed is shown as following:

> I need your help to generate some function calling datasets. I will provide you with a tool description, and you need to generate queries and corresponding answers based on this tool, i.e., the answers that call the tool to resolve the user's query. Here are my requirements:
>
> 1. For queries, try to use different vocabulary and syntax to ensure query diversity. Queries can be long or short, complex or concise. In short, try not to generate similar queries; I want to ensure query diversity.
> 2. The language of the queries should be as diverse as possible. This means a query can be a command, a question, or a request with detailed descriptions, etc.
> 3. The generated queries should cover all possible uses of the tool as much as possible, meaning the coverage of various parameters should be comprehensive, ensuring the tool can be used to complete various forms of work.
> 4. The generated queries should be solvable using the given tools.
> 5. For the queries you generate, you should provide answers using the tool, i.e., give the tool used and the values for each parameter.
> 6. When providing parameters, if a parameter has required=False, you may omit its value.
> 7. The generated data must be presented in the format given in my example.
> 8. The parameter values generated with function call generated must be values that can be inferred from the user's query; YOU CANNOT FABRICATE PARAMETERS THAT CANNOT BE OBTAINED FROM THE USER'S REQUEST.
> 9. Attach each answer with an id starting from 0. And if a tool should use the respone from another tool, you can reference it using #id, where id is the id of the tool.
>
> following are some examples:
> $examples
> Now I will give you a tool, and you help me generate 15 query-answer pairs.
> REMEMBER TO GENERATE THE RESULT IN JSON FORMAT LIKE THE EXAMPLE ABOVE REMEMBER NOT TO FABRICATE PARAMETERS FOR TOOLS. PARAMETERS SHOULD BE INFERED FROM USER QUERY.
> tool: $tool

In the prompt above, $examples will be replace by random samples sampled from xlam-function-calling-60k (Liu et al., 2024d). Below is an example:

```
tool: {
    "name: "...",
    "description": "...",
    "arguments": {
        ...
    }
}
response: {
    "query": "...",
    "answers": [
        {
            ...
        }
    ]
}
```

$tools will be replace by json formatted predefined function, below is an example:

```
tool: {
    "name": "ACTION_SET_ALARM",
    "description": "...".
    "arguments": {
        ...
    }
}
```

After seed generation stage, we will use another prompt to continuously generate data. Prompt is shown as following:

I need your help to generate some function calling datasets. I will provide you with a tool description and some example data for you. You need to generate queries and corresponding answers based on this tool, i.e., the answers that call the tool to resolve the user's query. Here are my requirements:

1. For queries, try to use different vocabulary and syntax to ensure query diversity. Queries can be long or short, complex or concise. In short, try not to generate similar queries; I want to ensure query diversity.
2. The language of the queries should be as diverse as possible. This means a query can be a command, a question, or a request with detailed descriptions, etc.
3. The generated queries should cover all possible uses of the tool as much as possible, meaning the coverage of various parameters should be comprehensive, ensuring the tool can be used to complete various forms of work.
4. The generated queries should be solvable using the given tools.
5. For the queries you generate, you should provide answers using the tool, i.e., give the tool used and the values for each parameter.
6. When providing parameters, if a parameter has required=False, it is not necessary to provide its value.
7. The query-answer pairs should cover as many possible uses of the tool as possible.
8. The generated data must be presented in the format given in my example.
9. The parameter values generated with function call generated must be values that can be inferred from the user's query; YOU CANNOT FABRICATE PARAMETERS THAT CANNOT BE OBTAINED FROM THE USER'S REQUEST.

following are tool I provided and some examples of query-answer pairs: tool: $tool examples: $examples

Now please help me generate 40 query-answer pairs. REMEMBER TO GENERATE THE RESULT IN JSON FORMAT LIKE THE EXAMPLE ABOVE REMEMBER NOT TO FABRICATE PARAMETERS FOR TOOLS. PARAMETERS SHOULD BE INFERED FROM USER QUERY.

$tool will be replaced by the json format of predefined functions shown early. $examples is the data sampled from the seed data generated previously.

When generating data of complex call, we slightly modify the prompt shown above. The seed generation prompt is shown below:

I need your help to generate some function calling datasets. I will provide you with a tool description,

and you need to generate queries and corresponding answers based on this tool, i.e., the answers that call the tool to resolve the user's query. Here are my requirements:

1. For queries, try to use different vocabulary and syntax to ensure query diversity. Queries can be long or short, complex or concise. In short, try not to generate similar queries; I want to ensure query diversity.
2. The language of the queries should be as diverse as possible. This means a query can be a command, a question, or a request with detailed descriptions, etc.
3. The generated queries should cover all possible uses of the tool as much as possible, meaning the coverage of various parameters should be comprehensive, ensuring the tool can be used to complete various forms of work.
4. The generated queries should be solvable using the given tools.
5. For the queries you generate, you should provide answers using the tool, i.e., give the tool used and the values for each parameter.
6. When providing parameters, if a parameter has required=False, you may omit its value.
7. The generated data must be presented in the format given in my example.
8. THE PARAMETER VALUES GENERATED WITH FUNCTION CALL GENERATED MUST BE VALUES THAT CAN BE INFERRED FROM THE USER'S QUERY; YOU CANNOT FABRICATE PARAMETERS THAT CANNOT BE OBTAINED FROM THE USER'S REQUEST.
9. THE GENERATED QUERY SHOULD CONTAIN ENOUGH INFOMATION SO THAT YOU COULD CORRECTLY GENERATE PARAMETER USED BY THE TOOLS. THIS IS ALSO TO GUARANTEE THAT YOU DON'T FABRICATE PARAMETERS.
10. You should use all the tools I provided to generate the query and answer. It means that you should generate a query that needs to use all the tools I provided to solve, and remember to provider an answer that uses all the tools to solve the query.
11. You can use the same tool multiple times in a single query to ensure the query diversity.
12. Attach each answer with an id starting from 0. And if a tool should use the respone from another tool, you can reference it using #id, where id is the id of the tool.
13. Generate data of nested function calls if possible. i.e., the argument of a function call is the response of another function call.

following are some examples:
$examples
Now I will give you a tool, and you help me generate 15 query-answer pairs. REMEMBER TO GENERATE THE RESULT IN JSON FORMAT LIKE THE EXAMPLE ABOVE AND PUT IT IN A JSON LIST.
REMEMBER YOU SHOULD USE ALL THE TOOLS AT ONE QUERY AND SOLVE IT WITH ALL TOOLS, AND GENERATE NESTED CALL IF POSSIBLE.
REMEMBER NOT TO FABRICATE PARAMETERS FOR TOOLS. PARAMETERS SHOULD BE INFERED FROM USER QUERY.
tools:
$tools

Prompt for continuously generating complex function calling data is:

I need your help to generate some function calling datasets. I will provide you with a tool description,

and you need to generate queries and corresponding answers based on this tool, i.e., the answers that call the tool to resolve the user's query. Here are my requirements:

1. For queries, try to use different vocabulary and syntax to ensure query diversity. Queries can be long or short, complex or concise. In short, try not to generate similar queries; I want to ensure query diversity.
2. The language of the queries should be as diverse as possible. This means a query can be a command, a question, or a request with detailed descriptions, etc.
3. The generated queries should cover all possible uses of the tool as much as possible, meaning the coverage of various parameters should be comprehensive, ensuring the tool can be used to complete various forms of work.
4. The generated queries should be solvable using the given tools.
5. For the queries you generate, you should provide answers using the tool, i.e., give the tool used and the values for each parameter.
6. When providing parameters, if a parameter has required=False, you may omit its value.
7. The generated data must be presented in the format given in my example.
8. THE PARAMETER VALUES GENERATED WITH FUNCTION CALL GENERATED MUST BE VALUES THAT CAN BE INFERRED FROM THE USER'S QUERY; YOU CANNOT FABRICATE PARAMETERS THAT CANNOT BE OBTAINED FROM THE USER'S REQUEST.
9. THE GENERATED QUERY SHOULD CONTAIN ENOUGH INFOMATION SO THAT YOU COULD CORRECTLY GENERATE PARAMETER USED BY THE TOOLS. THIS IS ALSO TO GUARANTEE THAT YOU DON'T FABRICATE PARAMETERS.
10. You should use all the tools I provided to generate the query and answer. It means that you should generate a query that needs to use all the tools I provided to solve, and remember to provider an answer that uses all the tools to solve the query.
11. You can use the same tool multiple times in a single query to ensure the query diversity.
12. Attach each answer with an id starting from 0. And if a tool should use the respone from another tool, you can reference it using #id, where id is the id of the tool.
13. Generate data of nested function calls if possible. i.e., the argument of a function call is the response of another function call.

Now I will give you some tools and some example data of query-answer pairs using these tools. Please help me generate 40 query-answer pairs. tools: $tools
examples: $examples

REMEMBER TO GENERATE THE RESULT IN JSON FORMAT LIKE THE EXAMPLE ABOVE AND PUT IT IN A JSON LIST.
REMEMBER YOU SHOULD USE ALL THE TOOLS AT ONE QUERY AND SOLVE IT WITH ALL TOOLS, AND GENERATE NESTED CALL IF POSSIBLE.
REMEMBER NOT TO FABRICATE PARAMETERS FOR TOOLS. PARAMETERS SHOULD BE INFERED FROM USER QUERY.

B Function Calling Prompts

In § 4.1, we've mentioned that we have tested 4 format of prompt: *json*, *code*, *json_short* and *code_short*. To unify our fine-tuning, we use chat to do function calling thus we only need to design the part of system, user and assistant using chat template.

In *json* or *code* format, the system prompt would be:

> You are an expert in composing functions. You are given a query and a set of possible functions. Based on the query, you will need to make one or more function calls to achieve the purpose. If none of the function can be used, point it out. If the given question lacks the parameters required by the function, also point it out. Remember you should not use functions that is not suitable for the query and only return the function call in tools call sections.

in *json_short* or *code_short* the system prompt would be:

> You are an expert in composing functions.

The user part of *json* or *code* is:

> Here is a list of functions that you can invoke:
> $functions
> Should you decide to return the function call(s), Put it in the format of
> $format_description
> $example
> If there is a way to achieve the purpose using the given functions, please provide the function call(s) in the above format. REMEMBER TO ONLY RETURN THE FUNCTION CALLS LIKE THE EXAMPLE ABOVE, NO OTHER INFORMATION SHOULD BE RETURNED.
> Now my query is: $user_query

$functions is the functions descriptions provided by retriever, in *code* or *code_short* format, it would be like:

```
Name:
    send_email
Description:
    Compose and send an email with optional attachments.

This function allows the user to compose an email with various options,
including multiple recipients, CC, BCC, and file attachments.
Args:
    to (List[str]): ...
    subject (str): ...
    ...
Returns:
    None
Example:
    # Send an email with a content URI attachment
send_email(
    to=["recipient@example.com"],
    subject="Document",
    body="Please find the attached document.",
    attachments=...
)
```

In *json* or *json_short*, functions would be describe directly in json format as shown in Listing 1.

$format_description in the prompt will be replace by detailed output format the model should follow. In *json* it will be:

```
[
```

```
    {
      "id": 0,
      "name": "func0",
      "arguments": {
          "arg1": "value1",
          "arg2": "value2",
          ...
      }
    },
    {
      "id": 1,
      "name": "func1",
      "arguments": {
          "arg1": "value1",
          "arg2": "value2",
          ...
      }
    },
    ...
]
If an argument is a response from a previous function call,
you can reference it in the following way like the argument
value of arg2 in func1:
[
    {
      "id": 0,
      "name": "func0",
      "arguments": {
          "arg1": "value1",
          "arg2": "value2",
          ...
      }
    },
    {
      "id": 1,
      "name": "func1",
      "arguments": {
          "arg1": "value1",
          "arg2": "#0",
          ...
      }
    },
    ...
]
This means that the value of arg2 in func1 is the return
value from func0 (#0 means the response from the function call with id 0).
```

In *code* format this will be

```
result1 = func0(arg1="value1", arg2="value2", ...)
```

```
result2 = func1(arg1="value1", arg2=result1, ...)
...
You can do nested function calling in the following way:
result1 = func0(arg1="value1", arg2="value2", ...)
result2 = func1(arg1="value1", arg2=result1, ...)
...
This means that the value of arg2 in func1 is the return value from func0.
```

$example in the prompt is used to test few-shot performance of a model.

The user prompt of *json_short* or *code_short* is much simpler withou *task instructions*:

```
Here is a list of functions: $functions
  Now my query is: $user_query
```

In *code* or *code_short* format the assistant output would be:

```
<sep>result1 = func0(arg1="value1", arg2="value2", ...)
result2 = func1(arg1="value1", arg2=result1, ...)</sep>
```

where $< sep >$ and $< /sep >$ can be any seperator set before fine-tuning.

In *json* or *json_short* format the assistant output would be:

```
[
    {
      "id": 0,
      "name": "func0",
      "arguments": {
          "arg1": "value1",
          "arg2": "value2",
          ...
      }
    },
    ...
]
```