DroidCall: A Dataset for LLM-powered Android Intent Invocation

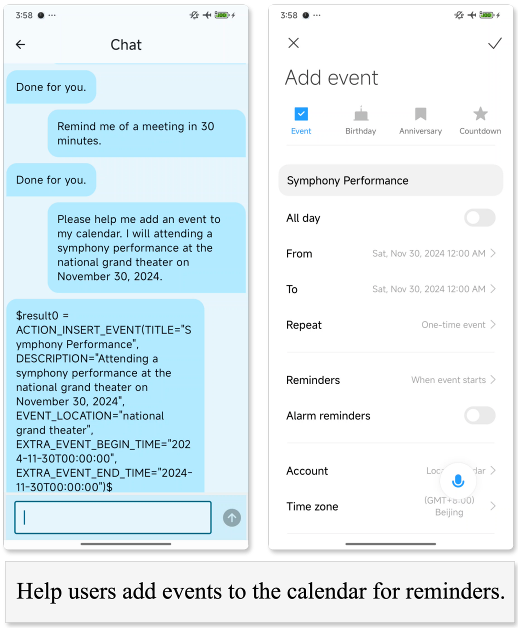
Anonymous ACL submission

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.



Done for you.

Remind me of a meeting in 30 minutes.

Done for you.

Please help me add an event to my calendar. I will attending a symphony performance at the national grand theater on November 30, 2024.

```
$result0 =
ACTION_INSERT_EVENT(TITLE="S
ymphony Performance",
DESCRIPTION="Attending a
symphony performance at the
national grand theater on
November 30, 2024",
EVENT_LOCATION="national
grand theater",
EXTRA_EVENT_BEGIN_TIME="202
4-11-30T00:00:00",
EXTRA_EVENT_END_TIME="2024-
11-30T00:00:00")$
```

Symphony Performance

All day

From Sat, Nov 30, 2024 12:00 AM

To Sat, Nov 30, 2024 12:00 AM

Repeat One-time event

Reminders When event starts

Alarm reminders

Account Loc dar

Time zone (GMT+8:00) Beijing

Help users add events to the calendar for reminders.

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., 2024c; 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.

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.

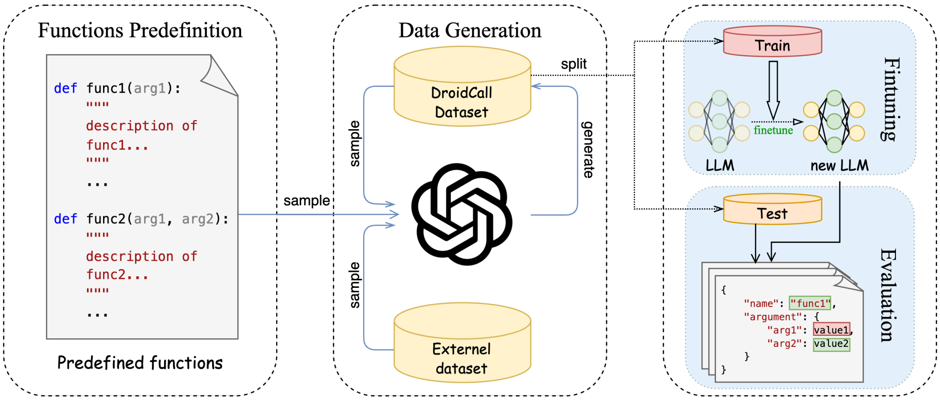
2.2 Mobile Device Control Agents

Agents for mobile device control have seen significant development. Early work (Venkatesh et al., 2022; Wang et al., 2023a; Wen et al., 2024) primarily focused on designing UI representations for models to understand mobile screens. With the advent of multimodal LLMs, agents like AppAgent (Yang et al., 2023b) and Mobile Agent (Wang et al., 2024b,a) now integrate visual capabilities to accomplish complex tasks 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 to the app.

2.3 LLMs for Function Calling

The emergence of LLMs has enabled powerful function-calling capabilities. Pioneering work like Toolformer (Schick et al., 2024) demonstrated LLMs' ability to use external tools. Developing these capabilities often requires substantial training data; following the Self-Instruct paradigm (Wang



```
def func1(arg1):
    """
    description of
    func1...
    """
    ...

def func2(arg1, arg2):
    """
    description of
    func2...
    """
    ...
```

```
{
    "name": "func1",
    "argument": {
        "arg1": value1,
        "arg2": value2
    }
}
```

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

et al., 2022), various efforts (Qin et al., 2023; Tang et al., 2023; Patil et al., 2023; Kim et al., 2023) have generated extensive function-calling datasets. Furthermore, numerous benchmarks and datasets, including APIGen (Liu et al., 2024d), Shortcuts-Bench (Shen et al., 2024a), ToolACE (Liu et al., 2024c), AppWorld (Trivedi et al., 2024), Android-World (Rawles et al., 2024), ToolSandbox (Lu et al., 2024b), and BFCLv3 (Yan et al., 2024), have been constructed to evaluate LLM tool use. AgentOhana (Zhang et al., 2024a) notably focused on standardizing data formats and training pipelines.

Our work introduces a reusable and customizable data generation pipeline specifically for Android intent invocation, aiming for better edge performance than large models like GPT-4o. We also provide straightforward methods for fine-tuning and evaluation. While similar mobile function-calling agents exist, such as TinyAgent (Erdogan et al., 2024) (macOS-specific) and Octopus (Chen and Li, 2024) (requires model architecture adjustments), neither offers publicly available code for data generation or fine-tuning, distinguishing our contribution.

3 `DroidCall` Dataset and Workflow

This section outlines the `DroidCall` framework, detailing its three key phases for building Android intent invocation capabilities: Function Predefinition, Data Generation, and Model Fine-tuning & Evaluation, as illustrated in Figure 2. Our data generation method requires minimal human supervision and is easily extensible. We conclude with an end-to-end demonstration of device control using fine-tuned LLMs.

3.1 Collecting Android Intents

In Android development, an intent is a messaging object facilitating communication between app components, used to request actions. Intents are broadly categorized: **Explicit Intents** target specific components (e.g., internal app communication), while **Implicit Intents** declare a general action, allowing any compatible component to respond.

`DroidCall` aims to enable models for function calling on Android devices for common operations. Implicit intents are particularly suitable for this, as they efficiently express user intentions and utilize system resources.

To construct the `DroidCall` dataset, we reviewed Android's official documentation (com, 2024). From this, we selected most frequently-used implicit intents, encapsulating them into 24 predefined functions. This selection covers the majority of standardized inter-app communication scenarios, enabling common operations such as 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. The process is orchestrated by a *Collector* component that coordinates interactions among other key components: the *Sampler*, *LLM*, and *Filter*. We first introduce these key components, and subsequently 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 the overall data generation workflow orchestrated by the Collector.

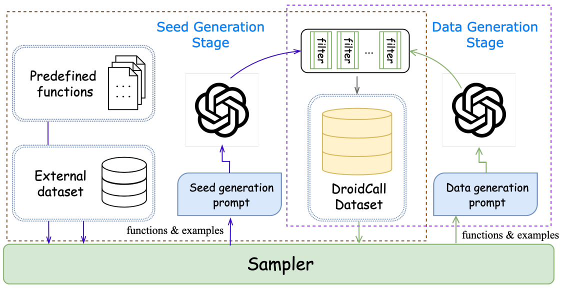


Figure 3: Details of data generation in `DroidCall`. To avoid manually creating seed data, `DroidCall` initially samples examples from an external dataset to generate its first set of data. Subsequently, the data is used as seed data to continuously generate new data, thereby eliminating the need for 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, a choice motivated by preliminary assessments of various models, as detailed in Appendix B.

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 serves as the central orchestrator of the data generation pipeline. It manages the workflow, directing the *Sampler* to retrieve source data, passing data and prompt templates to the *LLM* for raw output generation, routing the LLM's output through the sequence of *Filters* for validation and deduplication, and finally collecting the processed high-quality data.

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":"Help me call my friend Sophia.",
   "answers":[
      {
         "id":0,
         "name":"get_contact_info",
         "arguments":{
            "name":"Sophia",
            "key":"phone"
         }
      },
      {
         "id":1,
         "name":"dial",
         "arguments":{
            "phone_number":"#0"
         }
      }
   ]
}
```

Listing 4: Complex call example

The DroidCall dataset comprises 10K training(7589 simple entries and 2692 complex entries) and 200 test entries. The test set was randomly sampled from the generated data. This size was chosen to enable efficient evaluation cycles while facilitating meaningful comparison of model capabilities. As detailed in Appendix A, extensive analysis on test sets of varying sizes demonstrates the robustness of the conclusions drawn from evaluating on this test set.

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* (specifically, the functions relevant to the given task), 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 F.

3.4 Putting It All Together

We fine-tune SLMs for Android intent invocation using the DroidCall dataset. While our main evaluation (§4) focuses on the SLM's core mapping capability assuming relevant functions are provided as input, a real-world application necessitates dynamically identifying applicable functions from available tools. To demonstrate the feasibility of integrating our fine-tuned models into such an end-to-end system, we developed an Android application demo (Figure 4). The demo comprises two key components:

Retriever: In this end-to-end demonstration, the Retriever dynamically identifies relevant functions from the predefined set based on the user query. It leverages GTE (Li et al., 2023) for word embeddings and ObjectBox (obj, 2024) as the vector database, passing retrieved functions to the Intent Invocation Model.

Intent Invocation Model: This model takes the user query and retrieved functions as input, outputting the necessary function calls. For this, we use PhoneLM-1.5B (Yi et al., 2024) fine-tuned on the DroidCall dataset.

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 this end-to-end demo, where the fine-tuned model assists users in adding a calendar event.

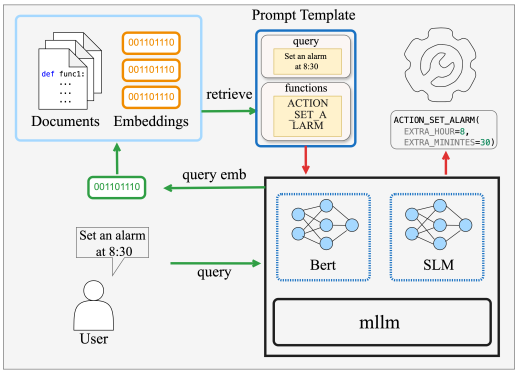


Figure 4: Design of our demo.

4 Experiments

We first explore the impact of prompt designs on model fine-tuning, identifying an optimal format. We then demonstrate DroidCall's superior performance over general function-calling datasets for Android intent invocation. Next, we analyze the contribution of different DroidCall data types through an ablation study. 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, requiring exact matches of function identity and all 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, averaged across all calls:

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

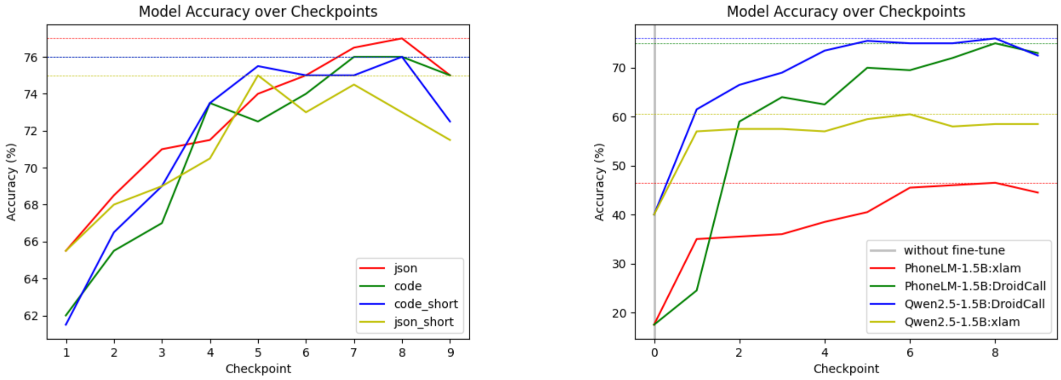
We evaluate SLMs using the 200 test entries from DroidCall. Following the input format used during training (§3.3), the prompts for evaluation include descriptions of the relevant functions required for each task. This setup allows us to isolate and measure the SLM's core ability to map user queries to correct function calls, decoupling the evaluation from the function retrieval process required in a full end-to-end system (demonstrated in §3.4).

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*,



(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.

and *task instructions*, with the output being a *specific representation for calling a function*. While the *user query* is user-provided, we designed the remaining components to evaluate their impact on fine-tuning performance.

json. Uses JSON for *available function descriptions* and function call representations due to 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. Removes *task instructions* from the *json* and *code* formats (*json_short* and *code_short*), hypothesizing they may be unnecessary after fine-tuning.

We fine-tuned Qwen2.5-1.5B-Instruct using four prompt formats. Figure 5(a) shows accuracy. The *code_short* format achieved comparable results to *json* with significantly fewer tokens (Table 1). We selected *code_short* for subsequent experiments.

4.2 Effectiveness of `DroidCall`

To verify `DroidCall` outperforms general datasets for Android intent invocation, we compared Qwen2.5-1.5B-Instruct and PhoneLM-1.5B fine-tuned on `DroidCall` and xlam-function-calling-60k (Liu et al., 2024d).

Both datasets were formatted using *code_short*. The xlam-function-calling-60k has 60k data points while `DroidCall` has 10k. To ensure equivalent training instances, we trained 4 epochs on xlam-function-calling-60k and 24 epochs on `DroidCall`. Results across 9 checkpoints (0th is pre-finetuning) are in Figure 5(b).

Accuracy improves with fine-tuning on both datasets, but the xlam-function-calling-60k model quickly plateaus. Improvement with `DroidCall` is substantially more significant.

This demonstrates that a task-specific dataset like `DroidCall` yields better results than a general-purpose dataset for a specific task. Furthermore, PhoneLM's performance, initially lower than Qwen's, reaches parity by the end of fine-tuning with `DroidCall`. This suggests `DroidCall` aids models in leveraging pre-trained knowledge for Android control, allowing models with comparable pre-trained knowledge to reach similar performance levels on this task.

4.3 Impact of Simple vs. Complex Data Types

The `DroidCall` dataset comprises 7589 simple data entries and 2682 complex entries(approx. 3:1). To understand their contribution and justify their ratio, we conducted an ablation study.

We fine-tuned Gemma-2-2b-it on subsets: simple only, complex only, and the full dataset. To account for subset sizes, we adjusted epochs (32 for simple, 96 for complex, 24 for combined) to equalize training steps. The best checkpoint for each was selected.

Results in Table 3 demonstrate the complementary value. Simple data alone achieves a solid baseline. Complex data alone yields lower accuracy, likely due to difficulty. The combined dataset

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		
GPT-4-turbo (2024-04-09)		78.5	91.2	**83.1**	**95.1**		

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.

achieves significantly higher accuracy. This shows simple examples help basic learning, while complex examples are crucial for handling diversity and compositionality, justifying the dataset's composition and ratio.

Type	**Soft Accuracy (%)**	**Accuracy (%)**
S	78.8	68.0
C	69.6	50.5
S + C	93.9	85.0

Table 3: Ablation study on the impact of data types on Gemma-2-2b-it performance. S denotes Simple and C denotes Complex.

4.4 Performance of Different SLMs

To test various edge-tailored SLMs and further verify DroidCall effectiveness, we tested Acc and Acc_{soft} under zero-shot, few-shot, and fine-tuned conditions. Zero-shot/few-shot used *json* prompts as *code_short* lacks task instructions needed for these settings; *json* was more effective than *code*.

Table 2 shows performance. Zero-shot results vary significantly, likely due to differing SFT and alignment effectiveness influencing instruction following. All models improve under few-shot conditions, suggesting better utilization of pre-trained knowledge.

After fine-tuning with DroidCall (*code_short* format), performance significantly improves. Inference requires only user query and function descriptions, greatly reducing prompt length compared to zero-shot/few-shot.

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.

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.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, et al. 2024b. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities. CoRR.

Zhenyan Lu, Xiang Li, Dongqi Cai, Rongjie Yi, Fangming Liu, Xiwen Zhang, Nicholas D Lane, and Mengwei Xu. 2024c. 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.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents. arXiv preprint arXiv:2405.14573.

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.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 16022–16076.

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.

Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Berkeley function calling leaderboard. https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html.

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 Evaluation Robustness Analysis

This appendix provides a detailed analysis addressing concerns regarding the size of our 200-sample test set relative to the 10K training set. While standard practice often suggests larger test sets to ensure robustness and reduce variance, our evaluation primarily aims to demonstrate the relative capabilities of fine-tuned Small Language Models (SLMs) compared to larger models for the specific task of Android intent invocation.

To investigate the reliability of evaluations conducted on test sets of this scale within the distribution of our generated data, we performed supplementary experiments using several capable large language models: Deepseek-chat, GPT-4o, and GPT-4o-mini. These models were evaluated in a zero-shot setting on randomly sampled test sets of varying sizes (200, 400, 600, 1000, 1500, and 2000 entries) drawn from our dataset. We chose these models to understand how performance metrics, particularly relative performance, behave across different evaluation scales on data structured for this domain. The results are presented in Table 4.

Model	Data Size	Zero-Shot Acc	Zero-Shot Acc$_{\text{soft}}$
deepseek-chat	200	75.0	89.9
	400	75.3	90.4
	600	71.1	86.3
	1000	73.3	87.5
	1500	78.5	91.2
	2000	71.5	86.0
GPT-4o	200	77.0	89.1
	400	74.3	87.9
	600	69.1	84.0
	1000	73.7	85.6
	1500	75.4	86.8
	2000	73.2	86.2
GPT-4o-mini	200	71.5	86.6
	400	74.8	88.5
	600	68.6	85.1
	1000	73.6	87.3
	1500	70.8	84.4
	2000	70.6	84.6

Table 4: Zero-shot Accuracy of Various Models on Randomly Sampled Test Sets of Different Sizes. The results demonstrate the stability of model performance across varying evaluation set sizes.

Analysis of Table 4 indicates that while there is some natural variation in absolute accuracy across different random test set samples, the relative performance ranking and general trends between these powerful models remain largely consistent across test set sizes ranging from 200 to 2000 samples. This stability in relative performance observed on diverse models within the DroidCall data distribution provides strong evidence that evaluating on a 200-sample set is sufficient for obtaining a representative signal for comparing the capabilities of different models in this specific intent invocation domain, which is the primary focus of our evaluation. Therefore, the conclusions drawn from our evaluations, including the comparison between our fine-tuned SLMs and larger models like GPT-4o, are supported by this analysis of evaluation robustness.

B Dataset Quality

This appendix provides a detailed discussion on the quality of the `DroidCall` dataset and the rationale behind selecting GPT-4-turbo as our primary data generation model. We address concerns regarding data reliability and the trade-offs involved in leveraging large language models for synthetic data creation.

B.1 Rationale for GPT-4-turbo in Data Generation

Our data generation pipeline, detailed in §3.2, primarily leveraged **GPT-4-turbo**. This decision stemmed from early observations during the initial experimental setup, where we informally assessed various LLMs, including GPT-4o, GPT-4o-mini, and DeepSeek, focusing on a practical balance of data quality, generation efficiency, and cost.

Key characteristics observed from our preliminary assessments:

- **GPT-4o-mini:** While potentially cost-effective, this model proved inefficient for generating usable data. It often produced a large volume of tokens that our filters discarded due to formatting errors or high redundancy, suggesting suboptimal output quality upon manual inspection.
- **GPT-4o:** This model performed significantly better than GPT-4o-mini but still exhibited slightly lower data quality compared to GPT-4-turbo.
- **DeepSeek:** DeepSeek consistently generated high-quality data. However, its API response time was considerably slower, making it impractical for the large-scale data generation required for `DroidCall`.

Ultimately, **GPT-4-turbo** emerged as the preferred choice due to its superior balance of output quality and generation speed, which was crucial for efficiently constructing our dataset. We acknowledge the potential of more recent powerful open-source models (like the latest Qwen versions) or other closed-source alternatives. However, a detailed comparative evaluation with these specific models was beyond the scope of this initial work due to resource constraints. We consider exploring their effectiveness for data generation an important direction for future research.

B.2 Ensuring Dataset Quality

To ensure the reliability and diversity of the `DroidCall` dataset, our generation pipeline incorporates several mechanisms:

- **Automated Filtering Pipeline:** As detailed in §3.2, all generated data undergoes a rigorous three-stage filtering process:
  - **JsonExtractor:** Ensures the output strictly adheres to the JSON format.
  - **FormatFilter:** Validates that the extracted JSON matches the predefined schema for function calls.
  - **SimilarityFilter:** Eliminates highly similar queries using LCS ROUGE scores (with an F-measure threshold of 75%), promoting data diversity.
- **Self-Instruct Paradigm:** By following the self-instruct paradigm (Wang et al., 2022), we leverage high-quality seed data (Liu et al., 2024d) to guide the LLM, ensuring that generated examples are relevant and structurally consistent with the target function-calling format. This approach minimizes manual annotation efforts while maintaining quality.
- **Function Predefinition:** The predefined set of 24 functions, based on Android Common Intents (com, 2024) (§3.1), provides a controlled and well-defined action space. This specificity reduces ambiguity in generated function calls and enhances the model's ability to learn precise mappings.

B.3 Human Evaluation of Generated Data

To provide a direct assessment of the generated data quality beyond automated metrics, we conducted a small-scale human evaluation. This evaluation aimed to validate both the technical correctness of the query-to-function mapping and the naturalness of the user queries.

Methodology: We randomly sampled 100 examples from the generated dataset. Three independent annotators were tasked with evaluating each example based on two criteria:

1. **Correctness:** A binary assessment of whether the query-to-function mapping was technically accurate (i.e., the function name and all its parameters perfectly matched the ground truth).
2. **Reasonableness:** A 5-point Likert scale (1: Very unreasonable, 5: Very reasonable) to assess the quality and naturalness of the task formulation in the user query.

Results: The results from the three annotators are summarized in Table ??.

Annotator	Correctness (%)	Reasonableness (avg)
1	90%	4.2
2	84%	3.8
3	86%	3.8

Table 5: Human Evaluation Results of Generated Data Quality (N=100)

The human evaluation results indicate a moderate to high level of quality in the generated data. Correctness rates, ranging from 84% to 90%, suggest that a significant majority of the generated examples accurately map user queries to their intended function calls. Reasonableness scores, averaging between 3.8 and 4.2, further support that the generated user queries are generally natural and logically consistent.

While this small-scale evaluation provides a strong indication of the dataset's reliability, we acknowledge that a more extensive assessment, particularly for complex multi-step examples, would further strengthen confidence. Our current findings, coupled with the robust automated filtering, establish a solid foundation for the dataset's use in fine-tuning SLMs for Android intent invocation.

C Predefined Function Specifications

This appendix documents the 24 core intent-based functions predefined within the DroidCall dataset. These functions encapsulate common Android operations via standardized implicit intents, serving as the foundational API for our framework. Detailed specifications are available at https://anonymous.4open.science/r/DroidCall-C100/api.py.

```
# Scheduling & Alarms
def ACTION_SET_ALARM(hour: int, minutes: int, message="", days=None): ...
def ACTION_SET_TIMER(duration: str, message=""): ...
def ACTION_SHOW_ALARMS(): ...
def ACTION_INSERT_EVENT(title: str, desc: str, location=None): ...

# Camera & Media
def ACTION_IMAGE_CAPTURE() -> str: ...  # Returns photo URI
def ACTION_VIDEO_CAPTURE() -> str: ...  # Returns video URI
def INTENT_ACTION_STILL_IMAGE_CAMERA(): ...
def INTENT_ACTION_VIDEO_CAMERA(): ...

# Contacts Management
def ACTION_PICK(data_type: str = "ALL") -> str: ...  # Returns contact URI
def get_contact_info(name: str, key: str) -> str: ...
def get_contact_info_from_uri(contact_uri: str, key: str)->str:
def ACTION_VIEW_CONTACT(uri: str): ...
def ACTION_EDIT_CONTACT(uri: str, info=None): ...
def ACTION_INSERT_CONTACT(info: dict): ...

# Messaging
def send_email(to: list, subject: str, body: str, cc=None, bcc=None): ...
def send_message(phone: str, subject: str, body: str): ...

# File Operations
def ACTION_GET_CONTENT(mime_type: str, multi=False) -> list: ...  # Returns URIs
def ACTION_OPEN_DOCUMENT(mime_types: list, multi=False) -> list: ...
def ACTION_CREATE_DOCUMENT(mime_type: str, name: str) -> str: ...  # Returns URI

# System Operations
def search_location(query: str): ...
```

```python
def dial(phone: str): ...
def web_search(query: str, engine="baidu"): ...
def open_settings(setting_type="general"): ...

# Utilities
def ACTION_GET_RINGTONE() -> str: ...  # Returns ringtone URI
```

Listing 5: Predefined Android Intent Functions

D Discussion on Function Selection, Coverage, and Scalability

This appendix provides additional context regarding the selection of the 24 predefined functions used in the DroidCall dataset and addresses related aspects of coverage and the method's scalability, building upon points raised during the review process.

D.1 Function Selection Rationale

Our approach focuses on enabling LLMs to invoke Android intents for common user operations. We manually defined a set of 24 functions based on a review of Android's official Common Intents documentation (com, 2024) and frequently performed tasks (e.g., setting alarms, sending emails, making calls). This curated set was chosen primarily to provide a controlled, representative, and practically relevant environment for evaluating the capability of fine-tuned SLMs in mapping diverse user instructions to specific, actionable intent calls. It allows us to demonstrate the core feasibility and performance gains of our data-driven approach within a defined scope.

D.2 Coverage Limitations

While the 24 functions cover a range of frequent user intents, they represent only a specific subset of the vast and dynamic Android ecosystem. The total number of potential intents is not fixed, as third-party applications can define custom intents, making a comprehensive quantification challenging. Therefore, our predefined set is not exhaustive and inherently limits the agent's capabilities to actions within this defined 24-function scope. This limitation is acknowledged in the main body's Limitations section.

D.3 Scalability and Generalizability

The manual predefinition of the action space directly impacts the scalability and generalizability of our method. Expanding the agent's capabilities to cover a significantly broader set of Android intents, or adapting the approach to other mobile platforms (e.g., iOS) or different interaction domains, would require substantial manual effort to define or map corresponding functions and their parameters. Our current approach reflects a deliberate trade-off, prioritizing the creation of a high-quality dataset and controllable evaluation environment for a specific, valuable task. In contrast, achieving broader scalability and generalizability would necessitate addressing the significant research challenges associated with automated action space discovery, intent parameter grounding, and data generation for unbounded or rapidly changing domains.

D.4 Future Directions

Future research directions aimed at mitigating these limitations include exploring automated or semi-automated techniques for discovering and defining intents and their parameters from various sources (e.g., documentation, UI analysis). Such approaches could help reduce manual effort, facilitate scaling to a wider range of Android functionalities, and potentially improve generalizability across different platforms and domains, complementing the data generation methodology presented in this work.

E 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.

F 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",
          ...
      }
    },
    ...
]
```