Grounding Code Generation with Input-Output Specifications

Anonymous Author(s)
Affiliation
Address
email

Abstract

Large language models (LLMs) have demonstrated significant potential in code generation. However, the code generated by these models occasionally deviates from the user's intended outcome, resulting in executable but incorrect code. To mitigate this issue, we propose GIFT4CODE, a novel approach for the instruction fine-tuning of LLMs specifically tailored for code generation. Our method leverages synthetic data produced by the LLM itself and utilizes execution-derived feedback as a key learning signal. This feedback, in the form of program input-output specifications, is provided to the LLM to facilitate fine-tuning. We evaluated our approach on two challenging data science benchmarks, ARCADE and DS-1000. Our results suggest that the method enhances the LLM's alignment with user intentions, reducing the incidence of executable but incorrect outputs.

1 Introduction

Large language models (LLMs) trained on code have demonstrated tremendous success as AI pair programmers in assisting developers writing code (Chen et al., 2021a; Austin et al., 2021; Li et al., 2023; Chowdhery et al., 2022; Li et al., 2022; Nijkamp et al., 2022; Fried et al., 2022; Li et al., 2023). Developers often interact with code LLMs using succinct natural language (NL) intents (*e.g.* $\boldsymbol{x}$ in Fig. 1) to describe their tasks (Barke et al., 2022; Ross et al., 2023). However, NL intents are often ambiguous (Yin et al., 2022b). This ambiguity can be problematic in complex tasks, such as manipulating Pandas DataFrames or PyTorch Tensors (Lai et al., 2022).

Auxiliary input-output (I/O) specifications, ranging from concrete I/O examples to high-level NL summaries (*e.g.* red text in Fig. 1), offer a natural way to reduce this ambiguity (Gulwani et al., 2015; Balog et al., 2016; Jain et al., 2022; Yin et al., 2022a). Prior to the emergence of LLMs, auxiliary specifications served as essential problem descriptions in program synthesis (Gulwani, 2016; Devlin et al., 2017; Shi et al., 2020). Real-world synthesis systems like FlashFill are testimony to the adoption and effectiveness of I/O specifications (Gulwani, 2011; Gulwani et al., 2012). In this work, we consider the problem of LLM-based code generation when the LLM has access to both a natural-language intent and an additional I/O specification.

However, code LLMs often fall short on following intents with additional complex semantic constraints like I/O specifications out-of-the-box, leading to plausible solutions that fail to satisfy the constraints (*e.g.* $\boldsymbol{y}'$, Fig. 1). Such a lack of *alignment* between the user's intent and the model's predictions (Chen et al., 2021a) could pose unnecessary burden on developers who are then required to fix the generated code (Bird et al., 2023). Therefore, we posit that addressing this misalignment by *grounding* the code generated by LLMs to the provided specifications is of paramount importance.

Instruction fine-tuning has emerged as an effective strategy to tackle the issue of misalignment (Wei et al., 2021; Sanh et al., 2021; Chung et al., 2022). Classical approaches for instruction tuning typically

Submitted to the Workshop on Instruction Tuning and Instruction Following at NeurIPS 2023. Do not distribute.

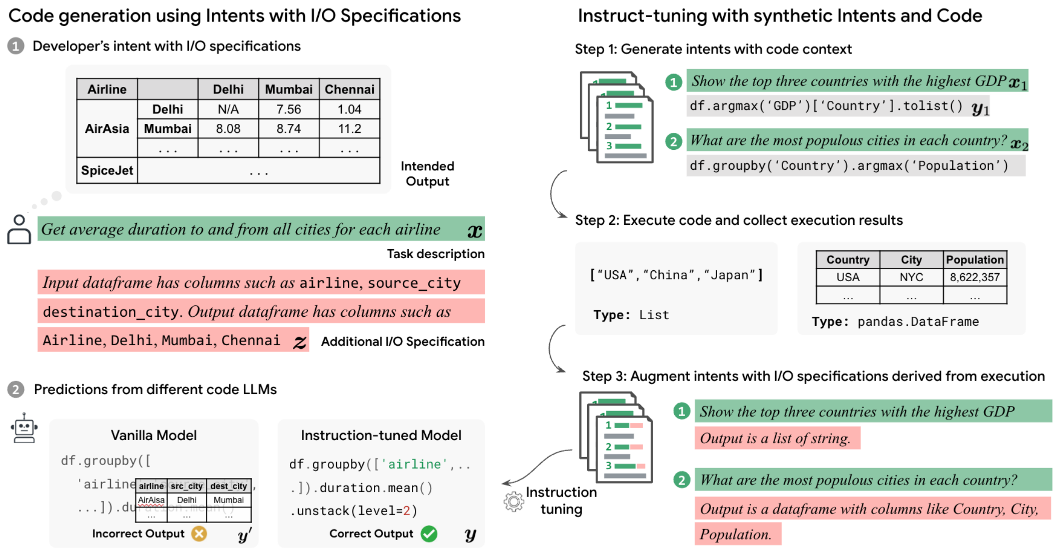

Figure 1: ***Left*:** Illustration of how developers prompt code LLMs with NL intents and I/O specifications to generate code with complex outputs (`pandas Dataframes`). Vanilla code LLMs fail to understand extra I/O specifications. ***Right*:** Our proposed instruction tuning approach uses synthetic intents and code solutions, where intents are augmented with I/O specifications derived from program execution results. Models trained on the synthetic data could better follow a developer's intent.

require a substantial amount of parallel labeled data of NL intents and gold model responses. The process of gathering such data is labor-intensive and time-consuming. Recent studies have suggested that generating synthetic instruction-following data using the LLM itself is a promising approach to improve alignment, with empirical success on natural language text generation tasks (Wang et al., 2022a; Honovich et al., 2022a; Taori et al., 2023; Peng et al., 2023, *inter alia*).

In this paper we build upon the recent success of instruction tuning using synthetic data and fine-tune code LLMs to follow NL intents with additional I/O specifications. Unlike existing approaches, our key insight is to leverage program execution for synthetic data generation. First, in contrast to other open-ended text generation tasks where assessing the quality of target responses is challenging, the quality of synthetic code generation data can be easily improved using heuristics such as code executability (Yin et al., 2022c). Moreover, from the program execution states one could derive precise and *aligned* I/O specifications that can be included in the intents to supervise a model to follow those extra semantic constraints (Fig. 1, *Right*). In other words, when fine-tuned on such synthetic data, a model learns to *ground* NL task descriptions to program execution states expressed as I/O specifications (Berant et al., 2013).

We apply our grounded instruction fine-tuning for code (GIFT4CODE) method to two challenging natural language to code generation applications: synthesizing complex `pandas` programs in computational notebooks (ARCADE, Yin et al. (2022b)) and answering data science questions on Stack Overflow (DS-1000, Lai et al. (2022)). First, we demonstrate the value of leveraging program execution information by showing that strong code LLMs can already be significantly improved by up to 10% absolute on ARCADE after fine-tuning on intents and executability-filtered code solutions *without* including any I/O specifications in synthetic data. Then, to further align model predictions to various types of user-provided I/O specifications, we derive those specifications at different levels of abstraction from code execution results. This ranges from concrete input/output examples to succinct natural language summaries of target variables (specifications in Fig. 1). By fine-tuning on parallel data of intents with I/O constraints and their target code solutions, the model is better at following a developer's intents while producing code that is more likely to execute to the desired outcome.

2 GIFT4CODE: Learning to Follow Intents with I/O Specifications

In this section we elaborate on GIFT4CODE, our proposed approach to fine-tune code LLMs to better follow developers' natural language intents along with I/O specifications, using synthetic parallel data. Fig. 1(*Right*) illustrates an overview of GIFT4CODE. We first synthesize a collection of intents with code solutions via few-shot prompting (§2.1), and then execute model-predicted code to derive I/O specifications from execution results (§2.2). Finally, we fine-tune the code LLM to predict code solutions given intents inlined with I/O specifications (§2.3).

2.1 Generating Synthetic Intents and Code Solutions

Programmatic Contexts We initialize a program state given some programmatic context and generate a series of contextualized NL-to-code problems for that context. As an example, the synthetic problems in Fig. 1 (*Right*) could have the contextual code `df = pd.read_csv("world_statistics.csv")`, which initializes the DataFrame variable `df`, subsequently used in the generated synthetic examples. The fact that our problems are contextualized sets our approach apart from existing instruction-tuning methods for text generation models (Wang et al., 2022a; Honovich et al., 2022a), where synthetic examples do not depend on any particular contexts. In our case, we mine those programmatic contexts from real-world code repositories, such as tabular datasets (e.g., `.csv`) used in data science notebooks on Github (§3).

Creating Initial NL Intents Given a programmatic context $\boldsymbol{c}$, we few-shot prompt an LLM to create a sequence of natural language intents $\{\boldsymbol{x}_i\}$ (*e.g.* $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ in Fig. 1(*Right*)). A problem $\boldsymbol{x}_i$ that appears later in the sequence might depend on the earlier ones $\{\boldsymbol{x}_{<i}\}$ (Nijkamp et al., 2022; Yin et al., 2022b). To generate NL intents, we use a "generalist" LLM instead of the code LLM that we aim to improve, since predicting intents conditioned on some context is similar to other text generation tasks, which could be better handled by a LM trained on general-purpose text data (Zelikman et al., 2022). The "generalist" LLM is a state-of-the-art general-purpose large language model. It achieves competitive results with GPT-4 on a variety of NL reasoning tasks. [1] Empirically, we observe that the problems generated by this LLM encompass a wide range of tasks relevant to the given programmatic context.

Predicting Code Solutions After generating an intent $\boldsymbol{x}$, we then prompt the code LLM to get a code solution $\boldsymbol{y}$ for $\boldsymbol{x}$ (*e.g.* $\boldsymbol{y}_1$ in Fig. 1(*Right*)). Specifically, a prompt to the LLM is the concatenation of the programmatic context $\boldsymbol{c}$ and the intent $\boldsymbol{x}$, with additional few-shot demonstrations of $\{\langle \boldsymbol{c}', \boldsymbol{x}', \boldsymbol{y}' \rangle\}$. Since many NL intents can be ambiguous and there could exist multiple alternative solutions (*e.g.* without additional I/O specifications, the intent in green in Fig. 1(*Left*) could be answered using tables with different layouts; see more in Yin et al. (2022b)), we therefore draw multiple candidate code solutions $\{\boldsymbol{y}\}$ for each intent. Intuitively, $\{\boldsymbol{y}\}$ could have a variety of alternative solutions for $\boldsymbol{x}$, each leading to different execution results. This equips the model with the capacity to predict code for the same task but with different user-provided I/O specifications.

2.2 Code Execution and Inference of I/O Specifications

Given the set of synthetic problems $\{\langle \boldsymbol{x}, \{\boldsymbol{y}\} \rangle\}$ generated by few-shot prompting, we execute the code for each problem (step 2, Fig. 1(*Right*)) and derive I/O specifications from the execution results as additional semantic constraints to be included in the intents (step 3, Fig. 1(*Right*)).

Specifically, for each candidate solution $\boldsymbol{y}$ of an intent, we first execute its original programmatic context $\boldsymbol{c}$, followed by executing $\boldsymbol{y}$. We trace the execution to collect the set of input and output variables in $\boldsymbol{y}$, denoted as $\{v\}$, which are used to derive I/O specifications (details below). Executing code with arbitrary programmatic contexts collected from the wild is highly non-trivial due to issues such as library dependency. However, the use of synthetic data alleviates the need for a complex environment setup.

Given the set of input and output variables extracted from execution results, we formulate an I/O specification, denoted as $\boldsymbol{z}$, which serves as additional information to augment a developer's intent, thereby providing a more comprehensive problem description. The level of detail and the style of these I/O specifications can vary based on the complexity of the problem and the developer's preferences. In this work, we investigate three distinct types of I/O specifications, each characterized by its own linguistic style and level of abstraction, as illustrated in Tab. 1.

[1] Details are publicly available in Anonymous (2023b). The model is now publicly available as an API, but was only privately accessible at the time of submission. Anonymized for double-blind review.

<table>
<tr><th>Spec. Type</th><th>Description</th><th>Example I/O Specification</th></tr>
<tr><td>TypeDesc</td><td>Variable type name</td><td>Generate a variable with name <code>df</code> and type <code>pandas.DataFrame</code></td></tr>
<tr><td>I/O Examples</td><td>Concrete I/O examples</td><td>Output variable <code>df</code>:
<code>|Bangalore(float)|Chennai(float)|Delhi(float)|Hyderabad (float)|Kolkata(float)|Hyderabad(float)|Kolkata(float)|...</code>
<code>|-----|-----|-----|-----|-----|-----|</code>
<code>| nan | 1.04 | 8.08 | 3.62 | 7.56 | 7.56 | 8.32 |</code>
<code>| 1.18 | nan | 11.96 | 6.80 | 6.31 | 8.75 |</code>
<code>| 8.46 | 11.10 | nan | 9.19 | 9.52 | 10.32 |</code></td></tr>
<tr><td>I/O Summary</td><td>LLM-generated NL summaries of I/O examples</td><td>Given the user intent and the code, the salient columns (at most given 3) in the input dataframe are airline, source_city, destination_city. The output dataframe has columns (at most given 3) such as Delhi, Mumbai, Chennai.</td></tr>
</table>

Table 1: Types of I/O specifications proposed in this work at different levels of abstraction. Example specifications are for the intent in Fig. 1(*Left*). Only the output specifications for I/O Examples are shown for brevity.

First, as a simple baseline, we utilize the variable type (TypeDesc, Tab. 1) as the I/O specification. Next, we incorporate the concrete values of the input/output variables into the specification, which we refer to as I/O Examples. This is reminiscent of classical program synthesis using I/O examples (Gulwani et al., 2012; Alur et al., 2013; Balog et al., 2016). However, in our scenario, these I/O examples are used in conjunction with natural language (NL) intents to define the problem, in line with Jain et al. (2022). Given that the majority of the problems in our synthetic dataset involve complex Python objects such as `pandas` `DataFrames`, we simplify the I/O specification to include only partial variable states (*e.g.* by excluding some rows and columns in large `DataFrames`).

In our effort to generate a more natural variety of I/O specifications that closely resemble the style of specifications in developers' NL intents, we employ an LLM to summarize the values of input/output variables $\{v\}$ into a succinct natural language description $\boldsymbol{z}$ (I/O Summary). Intuitively, the NL I/O summary includes salient information in the variables that can best clarify the original intent (*e.g.* the subset of columns in a `DataFrame` that are most relevant to solve a problem, as in Tab. 1, *Bottom*).

Specifically, we few-shot prompt the generalist LLM to generate $\boldsymbol{z}$, using information from its programmatic context $\boldsymbol{c}$, the intent $\boldsymbol{x}$, the code solution $\boldsymbol{y}$, as well as I/O variables $\{v\}$, *i.e.* $\boldsymbol{z} \sim P_{\text{LLM}}(\cdot \mid \boldsymbol{c}, \boldsymbol{x}, \boldsymbol{y}, \{v\})$. We then update the intent $\boldsymbol{x}$ by appending $\boldsymbol{z}$ to it. The few-shot exemplars used for prompting cover example I/O summaries for various types of Python objects, such as nested container types (*e.g.* nested `dicts`), along with more complex objects like `pandas` `DataFrames` and `pytorch` or `tensorflow` `Tensors`.

2.3 Fine-tuning Code LLMs to Follow Intents with I/O Specifications

Our approach, GIFT4CODE, aims to fine-tune code LLMs to generate code that adheres closely to the desired intents which are supplemented by I/O specifications. In our synthetic training data, each example $\langle \boldsymbol{c}, \boldsymbol{x}, \boldsymbol{y} \rangle$ consists of a programmatic context $\boldsymbol{c}$, an intent $\boldsymbol{x}$ augmented with I/O specifications, and the corresponding code solution $\boldsymbol{y}$. During fine-tuning, the code LLM learns to generate code that not only satisfies the provided intents but also respects the specified I/O constraints, while leveraging any relevant information in the programmatic contexts. In other words, we optimize $P_{\text{LLM}}(\boldsymbol{y} \mid \boldsymbol{c}, \boldsymbol{x})$. It is worth noting that the code LLM that undergoes this optimization is different from the "generalist" LLM employed to generate the NL intents and I/O specification $\boldsymbol{z}$.

3 Experiments

The core research question explored in this section is whether GIFT4CODE enhances the LLM's ability to follow developers' NL intents with ***complex*** I/O specifications. While common code generation benchmarks like HumanEval and MBPP Chen et al. (2021a); Austin et al. (2021) feature simple algorithmic tasks (e.g., sorting) utilizing basic Python data types (e.g., lists), thus allowing for the use of concrete I/O examples as specifications, they lack the diverse and complex I/O specifications that we aim to explore. For more open-ended tasks such as data science programming, the output data type is more complex and diverse (e.g., Pandas DataFrames, PyTorch tensors). Hence, we apply our method to two different data science code generation applications.

ARCADE (Yin et al., 2022b) is a benchmark of natural language to code generation in interactive data science notebooks. Each evaluation notebook consists of a series of interrelated NL-to-code problems in data wrangling (*e.g. "Min-max normalize numeric columns"*) and exploratory data analysis (*e.g.* intents in Fig. 1) using the `pandas` library. ARCADE features succinct NL intents to reflect the style of ephemeral queries from developers when prompting LLMs for code completion. More than 50% of the dataset's problems are under-specified, which means that additional I/O specifications could provide extra clarification. To construct programmatic contexts for synthetic training data generation, we scraped 7,500 CSV files that are used in public Jupyter notebooks. Each context contains a DataFrame import statement, for example, `df = pd.read_csv(·)`, followed by an NL description of the DataFrame to help the LLM understand its content. We generated 6 intents for each programmatic context and sampled 5 candidate code solutions for each intent. Roughly 60% of the code samples were executable. After filtering based on executability and API diversity (§2.1), we obtained around $20K$ synthetic training examples.

DS-1000 (Lai et al., 2022) is a benchmark of data science problems sourced from Stack Overflow (SO). Compared to ARCADE, problems in DS-1000 feature a wider variety of I/O types, such as `numpy`/`scipy` `Arrays` and `pytorch`/`tensorflow` `Tensors`, making it particularly appealing to evaluate our instruction tuning approach aimed at generating code following I/O specifications. However, in contrast to ARCADE which features succinct NL intents, DS-1000 follows the typical style of detailed problem descriptions found in SO posts. These elaborate descriptions often include additional information such as task background and descriptions of unsuccessful attempts, providing a more complex intent structure, with an average length of 140 words. Given that such elaborate intents may not reflect the style of developers' prompts to code LLMs, we do not focus on generating intents with similar styles. Instead, we held-out 500 problems in DS-1000 and use their annotated intents as training data, while evaluating on the remaining problems.[2]

3.1 Setup

Base Code LLM We use a strong decoder-only code language model with 62B parameters. The model was first pre-trained on a collection of 1.3T tokens of web documents and github code data, and was then fine-tuned on a disjoint set of 64B Python code tokens together with 10B tokens from Python Jupyter notebooks (Anonymous, 2023a).[3]

Learning Methods We evaluated the performance of both the baseline and instruction-tuned models across a range of data formats, as shown in Tab. 2. For each I/O specification type, we augmented the intents and few-shot exemplars with specifications of the corresponding type. Similarly, at test time, we augmented the intents with the same type of I/O specifications. The baseline models are tested under both zero-shot and few-shot prompting. For the latter, we manually created exemplars for all types of specifications. These exemplars were prepended to the prompt when querying the LLM for code generation during inference.

Simulate Noisy I/O Specifications at Test Time At testing time, the generation of I/O Summary underwent a minor modification from the process detailed in §2.2. We remove the concrete input/output variable states $\{v\}$ to produce noisy I/O summaries, simulating scenarios where users might give noisy I/O specifications (Devlin et al., 2017). While the "generalist" LLM uses the code solution to generate noisy I/O summaries, we remark that the code LLM, which we aim to evaluate, does not have access to the ground truth solution. In other words, the "generalist" LLM acts merely as a "data labeler" to create I/O summaries in prompts in order to construct the evaluation dataset. It is also a common practice in program synthesis to derive specifications from ground truth solutions, which then serve as the sole input to the model during its evaluation (Balog et al., 2016).

[2]We only use the annotated intents, while the code solutions and I/O specifications are still predicted by the LLM. We ensure the training and evaluation problems are disjoint and from different SO posts.

[3]Model details are available in Anonymous (2023a), but withheld from this submission for review.

Methods	ARCADE				DS-1000
	pass@5		*pass*@20		*pass*@1
	No Context	Full Context	No Context	Full Context	
	Zero-shot Prompting				
Code LLM (no spec.)	12.45	24.67	19.85	37.47	22.62
	Few-shot Prompting				
Code LLM (no spec.)	15.96	30.98	26.35	42.30	23.92
+ TypeDesc	16.58	29.68	29.68	42.30	25.90
+ I/O Examples	19.85	32.47	30.79	43.23	**26.41**
+ I/O Summary	**23.75**	**37.11**	**34.50**	**46.75**	26.25
	Synthetic Data Fine-tuning				
Code LLM (no spec.)	20.78	34.33	33.40	46.94	24.56
+ TypeDesc	21.52	36.73	33.58	48.61	27.35
+ I/O Examples	25.23	42.30	38.03	53.99	28.66
+ I/O Summary	**28.01**	**43.79**	**43.04**	**55.47**	**29.34**
StarCoder 15B	11.75	22.38	17.24	32.52	26.52
WizardCoder 15B	12.45	24.04	18.58	34.30	27.35

Table 2: *pass*@k on ARCADE and DS-1000. For each type of I/O specification in Tab. 1 (*e.g.* +I/O Summary), intents are augmented with I/O specifications of that type (*e.g.* intents inline with I/O summary) in fine-tuning data or few-shot exemplars. At test time, input intents use the same type of I/O specifications.

Metrics We adopted the *pass*@k metrics as defined in Chen et al. (2021a); Austin et al. (2021), which is calculated as the fraction of problems with at least one correct sample given k samples. Following Yin et al. (2022a), we drew 50 samples to calculate *pass*@5 and *pass*@20 to reduce the variance in ARCADE. Similar to Lai et al. (2022), we drew 40 samples to calculate *pass*@1 on DS-1000. Consistent with the original works' settings, the sampling temperature was set to 0.8 for ARCADE and to 0.2 for DS-1000 respectively.

3.2 Main Results

Tab. 2 presents the *pass*@k results on ARCADE and DS-1000. We evaluate both few-shot prompting and fine-tuning with synthetic data. Specifically, for ARCADE we evaluate on two versions of the dataset. First, we consider the original version where an intent is prefixed by prior notebook cells as its programmatic context (**Full Context**), as well as a **No Context** ablation to simulate the scenario where users query a code LLM using an intent without any context. This no-context setting is more challenging, where the zero-shot performance of the base code LLM is nearly halved. The standard errors in all cells of the table are less than 0.5%, and are excluded for clarity in presentation.

In our few-shot prompting experiments, we observe that *pass*@k generally improves with more detailed I/O specifications. Interestingly, on ARCADE, the improvements from prompting using I/O specifications compared to the baseline where no I/O specifications were used (**no spec**), are more notable in the more challenging no-context scenario (*e.g.* $15.96 \mapsto 23.75$ *v.s.* $30.98 \mapsto 37.11$ for +I/O Examples). This trend suggests that additional specifications could provide more valuable clarifications when adequate programmatic contexts are lacking.

Next, we fine-tune the base code LLM using our synthetic parallel data using different types of I/O specifications. Interestingly, without using any I/O specifications in the synthetic intents, on ARCADE the model already registers significant improvements compared to the zero- and few-shot settings. The model-predicted code solutions are filtered using executability heuristics, which helps improve the quality of the synthetic data, and a model fine-tuned on such data could generally be better at following users' intents, even without I/O specifications. Moreover, by fine-tuning the model to follow intents with additional I/O specifications, we observe significantly better results. We also remark that instruction fine-tuning using natural language I/O summaries (+I/O Summary) yields the best results on both datasets. Intuitively, those I/O summaries could encode salient information in target input and output variables through natural language descriptions, which could make it easier for the model to capture patterns in the data as compared to other more elaborate versions such as using concrete I/O examples.

We also evaluated Starcoder (Li et al., 2023) and its instruction tuned variant WizardCoder (Luo et al., 2023) on ARCADE and DS-1000. The result shows that GIFT4CODE is a more effective instruction tuning method in the data science domain. This is especially observed by the fact that GIFT4CODE offers much more relative improvement to the base model than the gains WizardCoder boasts over StarCoder. Overall, our results demonstrate that GIFT4CODE significantly improves the performance of code LLMs in following intents with I/O specifications at varying level of abstraction.

4 Related Work

Execution Guided Code Generation One area of study primarily focuses on utilizing execution as I/O examples, facilitating the synthesis of programs that align with the intended behavior. Gulwani (2016) involves synthesizing intended programs in an underlying domain-specific language (DSL) from example based specifications. This method has been further explored and adapted to different applications in subsequent studies (Devlin et al., 2017; Chen et al., 2018; Bunel et al., 2018). Another strand of research (Chen et al., 2021b; Wang et al., 2018; Ellis et al., 2019) leverages intermediate execution results to guide the search of programs. More recently, there have been attempts to utilize program execution results to verify and select code samples predicted by LLMs, either during auto-regressive decoding to prune search space (Zhang et al., 2023), or by few-shot prompting (Chen et al., 2023) and post-hoc reranking (Shi et al., 2022; Ni et al., 2023).

Instruction Fine-tuning Instruction fine-tuning is a widely adopted approach to address the misalignment issue in LLM-generated content. LLMs such as FLAN (Wei et al., 2021), which excel at understanding and executing instructions from prompts, are trained on labeled training data. Reinforcement learning with human feedback (RLHF) aims to mitigate the amount of labeling effort using model-based reward (Ouyang et al., 2022). Other works also confirmed the effectiveness of using instructional data in the fine-tuning stage (Mishra et al., 2021; Sanh et al., 2021; Chung et al., 2022; Wang et al., 2022b). To lower labeling cost, several recent works explored the possibility of automatic instruction generation (Ye et al., 2022; Zhou et al., 2022; Honovich et al., 2022b). In particular, SELF-INSTRUCT (Wang et al., 2022a) demonstrated that LLMs can be further improved by utilizing its own generation of instruction data. Our work differs from this line by considering execution-based specifications. Additionally, recent works attempted to distill instruction following data from more capable LLMs that have already been instruction-tuned (Honovich et al., 2022a; Taori et al., 2023; Chiang et al., 2023; Peng et al., 2023). In contrast, GIFT4CODE generates synthetic data from vanilla LLMs that have not gone through instruction-tunning.

Synthetic Data from LLMs Besides generating data for instruction following, a number of recent studies have also harnessed general-purpose LLMs to generate realistic synthetic data in areas where labeled data limited, such as language understanding and clinical research (Rosenbaum et al., 2022a; Tang et al., 2023; Borisov et al., 2022; Liu et al., 2022; Rosenbaum et al., 2022b; Josifoski et al., 2023). To improve the quality of synthetic data extracted from LLMs, such approaches usually apply a rejection sampling procedure and filter predictions based on domain-specific heuristics such as logical consistency (Bhagavatula et al., 2022; Yin et al., 2022c). GIFT4CODE is in spirit of this line in that it leverages program execution feedback to filter code predictions (Xu et al., 2020).

5 Conclusion

We have presented GIFT4CODE, a framework for instruction fine-tuning large language models of code in which the training is guided by execution based specifications. Empirically, we demonstrated how our approach enhances the quality of generated code, substantially improving accuracy on two challenging data science benchmarks, ARCADE and DS-1000.

References

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. *2013 Formal Methods in Computer-Aided Design*, pp. 1–8, 2013.

Anonymous. Anonymous code language model. 2023a.

Anonymous. Anonymous language model. 2023b.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *ArXiv*, abs/1611.01989, 2016.

Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *arXiv preprint arXiv:2206.15000*, 2022.

J. Berant, A. Chou, R. Frostig, and P. Liang. Semantic parsing on Freebase from question-answer pairs. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

Chandra Bhagavatula, Jena D. Hwang, Doug Downey, Ronan Le Bras, Ximing Lu, Keisuke Sakaguchi, Swabha Swayamdipta, Peter West, and Yejin Choi. I2d2: Inductive knowledge distillation with neurologic and self-imitation. *ArXiv*, abs/2212.09246, 2022.

Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools. *Queue*, 20(6):35–57, jan 2023. ISSN 1542-7730. doi: 10.1145/3582083. URL https://doi.org/10.1145/3582083.

Vadim Borisov, Kathrin Sessler, Tobias Leemann, Martin Pawelczyk, and Gjergji Kasneci. Language models are realistic tabular data generators. *ArXiv*, abs/2210.06280, 2022.

Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *ArXiv*, abs/1805.04276, 2018.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021a.

Xinyun Chen, Chang Liu, and Dawn Xiaodong Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

Xinyun Chen, Dawn Xiaodong Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. In *Neural Information Processing Systems*, 2021b.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *ArXiv*, abs/2304.05128, 2023.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam M. Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Benton C. Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz, Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311, 2022.

Hyung Won Chung, Le Hou, S. Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Wei Yu, Vincent Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed Huai hsin Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. *ArXiv*, abs/2210.11416, 2022.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *ArXiv*, abs/1703.07469, 2017.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Joshua B. Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *Neural Information Processing Systems*, 2019.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM-SIGACT Symposium on Principles of Programming Languages*, 2011. URL https://api.semanticscholar.org/CorpusID:886323.

Sumit Gulwani. Programming by examples - and its applications in data wrangling. In *Dependable Software Systems Engineering*, 2016.

Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55:97–105, 2012.

Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. Unnatural instructions: Tuning language models with (almost) no human labor. *ArXiv*, abs/2212.09689, 2022a.

Or Honovich, Uri Shaham, Samuel R. Bowman, and Omer Levy. Instruction induction: From few examples to natural language task descriptions. *ArXiv*, abs/2205.10782, 2022b.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 1219–1231, 2022.

Martin Josifoski, Marija Sakota, Maxime Peyrard, and Robert West. Exploiting asymmetry for synthetic training data generation: Synthie and the case of information extraction. *ArXiv*, abs/2303.04132, 2023.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501, 2022.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nourhan Fahmy, Urvashi Bhattacharyya, W. Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jana Ebert, Tri Dao, Mayank Mishra, Alexander Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean M. Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! 2023.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom, Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de, Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey, Cherepanov, James Molloy, Daniel Jaymin Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de, Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378:1092 – 1097, 2022.

Qi Liu, Zihuiwen Ye, Tao Yu, Phil Blunsom, and Linfeng Song. Augmenting multi-turn text-to-sql datasets with self-play. In *Conference on Empirical Methods in Natural Language Processing*, 2022.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *ArXiv*, abs/2306.08568, 2023. URL https://api.semanticscholar.org/CorpusID:259164815.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. In *Annual Meeting of the Association for Computational Linguistics*, 2021.

Ansong Ni, Srini Iyer, Dragomir R. Radev, Ves Stoyanov, Wen tau Yih, Sida I. Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. *ArXiv*, abs/2302.08468, 2023.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Haiquan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. 2022.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155, 2022.

Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. Instruction tuning with gpt-4. *ArXiv*, abs/2304.03277, 2023.

Andrew Rosenbaum, Saleh Soltan, Wael Hamza, Amir Saffari, Macro Damonte, and Isabel Groves. Clasp: Few-shot cross-lingual data augmentation for semantic parsing. In *AACL*, 2022a.

Andrew Rosenbaum, Saleh Soltan, Wael Hamza, Yannick Versley, and Markus Boese. Linguist: Language model instruction tuning to generate annotated utterances for intent classification and slot tagging. In *International Conference on Computational Linguistics*, 2022b.

Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael J. Muller, and Justin D. Weisz. The programmer's assistant: Conversational interaction with a large language model for software development. *Proceedings of the 28th International Conference on Intelligent User Interfaces*, 2023.

Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal V. Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Févry, Jason Alan Fries, Ryan Teehan, Stella Rose Biderman, Leo Gao, Tali Bers, Thomas Wolf, and Alexander M. Rush. Multitask prompted training enables zero-shot task generalization. *ArXiv*, abs/2110.08207, 2021.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural language to code translation with execution. *ArXiv*, abs/2204.11454, 2022.

Kensen Shi, David Bieber, and Rishabh Singh. Tf-coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44:1 – 36, 2020. URL https://api.semanticscholar.org/CorpusID:214605958.

Ruixiang Tang, Xiaotian Han, Xiaoqian Jiang, and Xia Hu. Does synthetic data generation of llms help clinical text mining? *ArXiv*, abs/2303.04360, 2023.

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

Chenglong Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov, and Rishabh Singh. Robust text-to-sql generation with execution-guided decoding. *arXiv: Computation and Language*, 2018.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language model with self generated instructions. *ArXiv*, abs/2212.10560, 2022a.

Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Anjana Arunkumar, Arjun Ashok, Arut Selvan Dhanasekaran, Atharva Naik, David Stap, Eshaan Pathak, Giannis Karamanolakis, Haizhi Gary Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krima Doshi, Maitreya Patel, Kuntal Kumar Pal, M. Moradshahi, Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkit Verma, Ravsehaj Singh Puri, Rushang Karia, Shailaja Keyur Sampat, Savan Doshi, Siddharth Deepak Mishra, Sujan Reddy, Sumanta Patro, Tanay Dixit, Xudong Shen, Chitta Baral, Yejin Choi, Noah A. Smith, Hanna Hajishirzi, and Daniel Khashabi. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. In *Conference on Empirical Methods in Natural Language Processing*, 2022b.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners. *ArXiv*, abs/2109.01652, 2021.

Silei Xu, Sina J. Semnani, Giovanni Campagna, and Monica S. Lam. Autoqa: From databases to q&a semantic parsers with only synthetic training data. *ArXiv*, abs/2010.04806, 2020.

Seonghyeon Ye, Doyoung Kim, Joel Jang, Joongbo Shin, and Minjoon Seo. Guess the instruction! flipped learning makes language models stronger zero-shot learners. *ArXiv*, abs/2210.02969, 2022.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, A. Eashaan Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. Natural language to code generation in interactive data science notebooks. *ArXiv*, abs/2212.09248, 2022a.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. Natural language to code generation in interactive data science notebooks. 2022b.

Pengcheng Yin, John Frederick Wieting, Avirup Sil, and Graham Neubig. On the ingredients of an effective zero-shot semantic parser. In *Annual Conference of the Association for Computational Linguistics (ACL)*, Dublin, Ireland, May 2022c. URL https://arxiv.org/abs/2110.08381.

E. Zelikman, Qian Huang, Gabriel Poesia, Noah D. Goodman, and Nick Haber. Parsel: A (de-)compositional framework for algorithmic reasoning with language models. 2022.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *ArXiv*, abs/2303.05510, 2023.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *ArXiv*, abs/2211.01910, 2022.