Plan with Code: Comparing approaches for robust NL to DSL generation

Nastaran Bassamzadeh **Chhaya Methani**
Microsoft Corporation
Redmond, USA

Abstract

Planning in code is considered a more reliable approach for many orchestration tasks. This is because code is more tractable than steps generated via Natural Language and make it easy to support more complex sequences by abstracting deterministic logic into functions. It also allows spotting issues with incorrect function names with the help of parsing checks that can be run on code. Progress in Code Generation methodologies, however, remains limited to general-purpose languages like C, C++, and Python. LLMs continue to face challenges with custom function names in Domain Specific Languages or DSLs, leading to higher hallucination rates and syntax errors. This is more common for custom function names, that are typically part of the plan. Moreover, keeping LLMs up-to-date with newer function names is an issue. This poses a challenge for scenarios like task planning over a large number of APIs, since the plan is represented as a DSL having custom API names. In this paper, we focus on workflow automation in RPA (Robotic Process Automation) domain as a special case of task planning. We present optimizations for using Retrieval Augmented Generation (or RAG) with LLMs for DSL generation along with an ablation study comparing these strategies with a fine-tuned model. Our results showed that the fine-tuned model scored the best on code similarity metric. However, with our optimizations, RAG approach is able to match the quality for in-domain API names in the test set. Additionally, it offers significant advantage for out-of-domain or unseen API names, outperforming Fine-Tuned model on similarity metric by 7 pts.

1 Introduction

There has been significant progress made in improving and quantifying the quality of Natural Language to Code Generation or NL2Code (Chen et al., 2021, Nguyen and Nadi, 2022). Recent improvements in models for general-purpose languages like Python, C++ and Java can be attributed to larger LLMs (ChatGPT, GPT-4) and the availability of Pre-trained open-source models (Code Llama, Abdin et al., 2024, Codestral) advancing the state-of-the-art. However, there hasn't been a focus on improving quality of Natural Language to Domain Specific Languages or NL2DSL, which a lot of enterprise applications rely on.

Domain Specific Languages (or DSLs) are custom computer languages designed and optimized for specific applications. Examples of DSLs include SQL and industry-specific languages for formalizing API calls, often using formats like JSON or YAML to represent API sequences. In this paper, we focus on the task of generating a DSL used for authoring high-level automation workflows across thousands of web-scale APIs. These workflows support a variety of customer scenarios like invoice processing, sales lead integration with forms/emails etc. The automation DSL represents API names as functions and codifies a sequence of API calls (or a plan) along with conditional logic over the invocation of APIs. The syntax itself borrows from known languages like Javascript, however, the logic resembling the workflow along with the custom function names, make it unique. An example of the DSL is shown in Figure 1.

Existing code generation methods are hard to adapt for this scenario due to the frequent hallucinations and syntax errors. This is largely due to the custom API names, high cardinality and diversity of APIs in public as well private domain along with the ever-changing API landscape. A typical workflow can choose among thousands of publicly available APIs (eg. Database connectors, Emails, Planner, Notifications etc.) as well as private APIs in tenant subscriptions and string them together to automate business processes.

In this paper, we outline an end to end system architecture for NL2DSL generation with high response rate using selective improvements to RAG

techniques (Liu et al., 2023, Poesia et al., 2022) using OpenAI models. We focus on bench-marking the impact of different contexts used for grounding. We fine-tuned a Codex model for NL2DSL and show a comparative analysis of the impact of the approaches used to optimize RAG.

The remainder of this study is structured as follows. In Section 2, we present the NL2DSL problem formulation along with literature review. Section 3 lays out and describes the optimizations we made to RAG as discussed above along with the benchmark Fine-Tuned model. Section 4 discusses Data Generation, Metric definition and Section 5 shares our results and discussion followed by Conclusion in Section 6.

2 Related Work

2.1 Code Generation or Program Synthesis

Program Synthesis is a hard research problem (Jain et al., 2021, Li et al., 2022, Xu et al., 2021). It has gained significant interest with many open-source models (Code Llama, Li et al., 2023, Codestral, Abdin et al., 2024, Chen et al., 2021) focusing on general programming languages. Many of these advancements have been achieved through pre-training language models for code generation with a focus on improving datasets (Code Llama, Abdin et al., 2024). However, for domain adaptation, **instruction fine-tuning** on top of a base model remains a popular approach (Chen et al., 2021, Gao et al., 2023, Lewkowycz et al., 2022, Patil et al., 2023).

Prompting LLMs is an alternative technique for code generation (Liu et al., 2023, White et al., 2023, Wei et al., 2023, Kojima et al., 2023). Most papers focus on metaprompt optimization and learning, while Poesia et al., 2022 focused on improving response quality by dynamically selecting few-shots for grounding the model.

2.2 Reasoning and Tool Integration

For modeling the problem of selecting a sequence of API calls, we need to consider formulating it as a **planning** or **reasoning** task. LLMs show remarkable reasoning capability, however, they also have limitations when it comes to staying up-to-date with recent knowledge, performing mathematical calculations etc. A popular way to overcome this has been granting the LLMs access to external tools. This framework gained significant popularity with OpenAI Code Interpreter's success (OpenAI Code Interpretor).

External Tool Integration has been studied since with a focus on including specific tools such as web search (Schick et al., 2023), python code interpreters (Gao et al., 2023, OpenAI Code Interpretor), adding calculators (Parisi et al., 2022 Gao et al., 2023) and so on. Expanding the tool set to a generic list of tools has been explored (Schick et al., 2023, Patil et al., 2023), but it remains limited and often predicts single tool instead of sequences needed for most enterprise scenarios. Tool Use has mostly been explored in the context of generating more accurate text outputs for Q&A tasks with the help of external tools(Schick et al., 2023, Parisi et al., 2022).

There is an increase in focus on incorporating LLM's code generation capabilities to reasoning and task orchestration making this an area of active research (Gao et al., 2023, Liang et al., 2023, Patil et al., 2023). However, similar to Q&A scenarios mentioned above, most of the research either limits the tools to a set of small well-documented APIs (Gao et al., 2023, Liang et al., 2023), or limited their scope to predicting a single output API (Patil et al., 2023, Schick et al., 2023).

Posing the reasoning or orchestration task as a code generation problem is similar to the API sequence generation scenario highlighted in this paper. Improving the quality of Natural Language to DSL generation, is thus beneficial for both reasoning and plan generation.

2.3 Contributions

NL2DSL generation suffers from the hallucination and quality issues we discussed in Section 1. Few studies address the challenges of end-to-end DSL generation, specifically over a large set of custom APIs. In this paper, we present an end-to-end system architecture with improved strategies to add grounding context with known RAG techniques. We also present an ablation study showing improvements for DSL generation quality for enterprise settings. DSL samples in our test set consider API or tool selection sequences of 5-6 API calls, also referred to as chain of tools, which is a first to the best of our knowledge. We also consider the real-world scenarios of adding conditional logic with API calls as shown with an example in Figure 1.

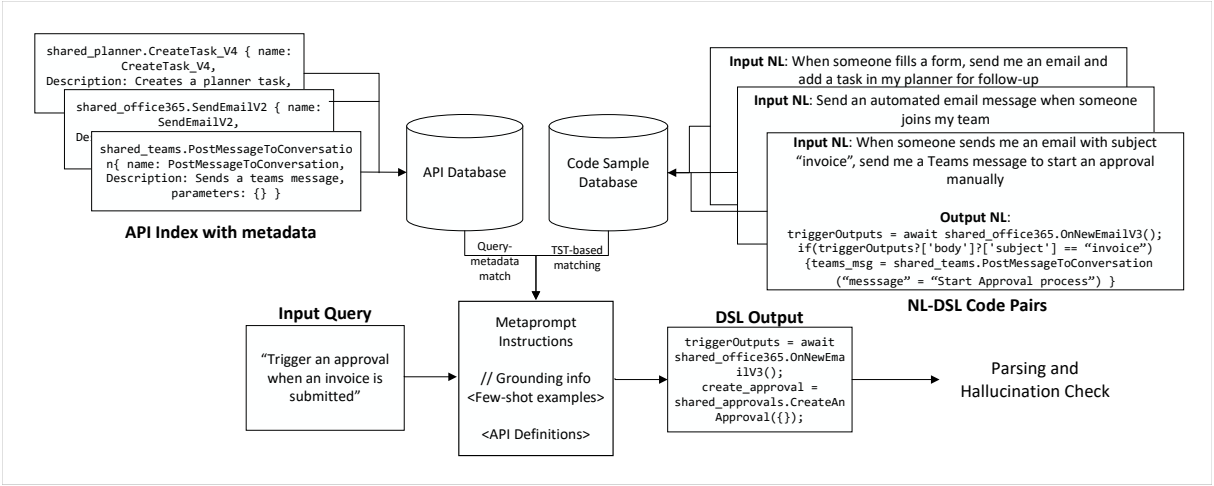

Figure 1: System Architecture to show e2e working of our DSL generation methodology using RAG. TST based semantic mapping retrieves the relevant code snippet as shown. This helps get the right syntax. However, it gets the correct function name for approval from the API metadata

3 Methodology

3.1 Fine-Tuned NL2DSL Generation Model

We took the Codex base model from OpenAI due to it's pre-training with code samples and used LoRA-based fine-tuning approach. The training set consists of NL-DSL pairs, NL refers to the user query and the DSL represents the workflow that the user is looking to automate. The training set consists of a pool of 67k samples in the form of (prompt, flow) tuples with the NL generated synthetically (details in Section 4.1, examples of NL-DSL are shared in Figure 1 and Appendix 8).

We ran many iterations on this model to improve performance on the test set, specifically for the body and tail connectors, and went through multiple rounds of data augmentation. We found that predicting the parameter keys was very challenging with the fine-tuned model due to limitation of high-quality data generation.

3.2 Grounding with dynamically selected few-shots

We tried two types of grounding information for RAG based DSL generation as described below. For each technique, we selected 5 and 20 few-shots dynamically, and compared performance impact driven by the approach used for sample selection.

3.2.1 Pre-trained Model

The first approach is using a vanilla Pre-trained model for determining the semantic similarity of NL-DSL samples based on the NL query. We computed the embeddings of NL queries using a DistilRoBERTa Pre-trained model. We created a Faiss Index (Douze et al., 2024) for these embeddings to help with search over the dense embedding space.

3.2.2 TST based BERT Fine-tuning

In this approach, we fine-tuned the Pre-trained model to improve retrieval accuracy of few-shots similar to the work done by Poesia et al., 2022.

To get positive and negative samples for fine-tuning, we generated embeddings for all NL queries in our dataset using a Pre-trained Tansformer model. A pair of tuples is considered a positive sample if the cosine similarity between their corresponding NL prompts is greater than 0.7 and negative otherwise. We generated $100k$ pairs this way and leveraged them as training data for our fine-tuned model.

The loss function used by TST (Equation 1 from Poesia et al., 2022) is minimizing the Mean-Squared Error between the vanilla loss functions comparing the utterances (u_i, u_j) and the target programs (p_i, p_j). Program similarity is denoted by S. We used a Jaccard score over lists of API function names as the similarity metric between programs.

$$L_{TST}(\theta) := E_{i,j\ D}[f_\theta(u_i, u_j) - S(P_i, p_j)]^2 \quad (1)$$

3.3 Grounding with API Metadata

In addition to few-shots, we appended the API metadata in the metaprompt. This metadata includes Function Description along with the parameter keys and their description (Example API Function Definition shared in Appendix 8). We followed

the below two approaches for selecting the metadata to be added.

3.3.1 API Function Definitions for Few-Shots

For few-shot samples selected using the methods described above, we extracted the metadata for **each of** the functions present in those samples. This means that for the n few-shot samples dynamically added to the metaprompt, we iterated over all the API function names in each of these flows and added their function definitions to the metaprompt.

3.3.2 Semantic Function Definitions

Another approach for selecting function definitions is to retrieve semantically similar functions from a vector database created with API metadata. This approach is similar to the one followed by (LlamaIndex). We created an index of all API definitions and used the input NL query for search. Please note that this is different from the faiss index created for few-shot samples in Section 3.2.

We call this approach **Semantic Function Definition (SFD)** and will compare it with the **Regular FDs** described above. This approach can be specifically useful for tail-ish prompts where no few-shots might be retrieved.

4 Experiment Design and Metrics Definition

In this section, we outline the process of Dataset Generation and introduce the metrics we used for estimating the code quality. We then describe our experiments. We have used Azure AML pipelines and GPT-4 (16k token limit) for our experiments.

4.1 Dataset Generation

Our train and test set consists of a total of 67k and 1k samples, respectively. These samples are (prompt, flow) pairs with the workflows being created by users across a large set of APIs. We scrubbed PII from these automations and sampled workflows containing 700 publicly available APIs. We synthetically generated the corresponding Natural Language prompts using GPT-4.

4.2 DSL Generation Quality Metrics

We defined 3 key metrics to focus on code generation quality as well as syntactic accuracy and hallucination rate.

4.2.1 Average Similarity

Average Similarity measures the similarity between predicted flow and the ground truth flow and is defined using the Longest Common Subsequence match (LCSS) metric. Each flow is reduced to a list of API sequences and then the LCSS is computed. The final metric is reported as an average over all test samples. Hallucination and Parser failures lead to the sample being discarded and is assigned a similarity score of 0.

$$\text{Similarity} = \frac{\text{LCSS}(A, B)}{max(|\text{Actions}_A|, |\text{Actions}_B|)} \quad (2)$$

where $|\text{Actions}_A|$ is the number of actions in flow A and $|\text{Actions}_B|$ is the number of actions in flow B.

4.2.2 Unparsed rate

This metric captures the rate of syntactic errors. A flow that cannot be parsed by the parser is considered not usable for the purpose of similarity metric computation. Unparsed rate is computed as follow:

$$\%\text{unparsed flows} = \frac{|\text{Flows}_{\text{unparsed}}|}{|\text{Flows}_{\text{total}}|} \quad (3)$$

where, $|\text{Flows}_{\text{unparsed}}|$ is the number of flows that were not parsed and $|\text{Flows}_{\text{total}}|$is the total number of flows in the sample set.

4.2.3 Hallucination rate

This metric captures the rate of made-up APIs and made-up parameter keys in the generated code. Predicting a flow with a hallucinated API name is counted as a failure and leads to the code being considered invalid. However, hallucinated parameter keys would only lead to run-time errors which can be fixed down the line. Fixing these run-time errors is beyond the scope of this paper.

$$\%\text{made} - \text{up APIs} = \frac{|\text{Flows}_h|}{|\text{Flows}_{\text{parsed}}|} * 100 \quad (4)$$

$$\%\text{made} - \text{up parameters} = \frac{|\text{Flows}_{hp}|}{|\text{Flows}_{\text{parsed}}|} * 100 \quad (5)$$

where $|\text{Flows}_h|$ is the number of flows with hallucinated API names, $|\text{Flows}_{hp}|$ is the number of flows with hallucinated parameter key names and $|\text{Flows}_{\text{parsed}}|$ is the number of flows that were parsed correctly.

Model	Num. of few-shots	Avg. similarity	%Unparsed flows	%Made-up API names	%Made-up parameters
Pre-trained wo FD	20	**+0.03**	**−3.37**	−7.34	**−15.17**
TST wo FD	5	+0.02	−0.61	−3.53	−1.04
TST wo FD	20	**+0.03**	−2.85	**−8.49**	−14.58

Table 1: Impact of selecting **5 vs 20 few-shot** samples for TST and Pre-trained Model without adding API Function Definitions using GPT-4. The baseline uses Pre-trained Transformer Model with 5 few-shot samples.

Model	Avg. Similarity	%Unparsed flows	%Made-up API names	%Made-up API parameters
Pre-trained + FD	0	+2.75	−4.3	**−20.16**
TST wo FD	**+0.02**	**−0.61**	−3.53	−1.04
TST + FD	**+0.02**	+0.68	**−6.29**	−19.99

Table 2: Impact of selecting **5 few-shot** samples using TST vs. Pre-trained Model with and without API Function Definitions using GPT4 model. The baseline uses Pre-trained Transformer Model without API Function Definitions.

5 Results

In this section, we present the results of the above approaches on a test set of 1000 NL-DSL pairs. The test set is split in 864 in-domain samples and 136 out-of-domain samples. The NL component in these samples, while generated synthetically, has been evaluated by human judges for quality. They were also sampled to represent the distribution of APIs in actual product usage.

We compare the impact of each ablation in sections below. Please note that in the following sections all the results are presented as Δ change compared to a baseline scenario where the higher Δ is better for Avg. similarity and lower Δ is better for the rest of metrics capturing failures.

5.1 Impact of number of few-shots

We compare the impact of number of code samples added to the meta prompt with two different settings i.e. 5 few-shots vs 20 few-shots. We measured the results for both Pre-trained model as well as TST model. Results are shared in Table 1.

Looking at rows 2 and 4 having 20 few-shot samples, we can see that adding more few-shots improves the performance of both the Pre-trained as well as the TST model on all metrics. The gain is particularly pronounced for reducing the number of made-up API names as well as reducing the number of made-up API parameter keys.

5.2 TST vs Pre-trained Model

To compare the impact of selecting samples using TST vs Pre-trained model, we look at the impact with and without the presence of API Function Definitions (see Table 2 and Table 3). Here, we have used GPT4 model with 5 and 20 few-shots, respectively. TST with FD setting performs overall better than all other options with values close to the best in every metric.

This leads us to conclude that the presence of few-shot examples is supported by adding their API functions definitions (as described in Section 3). The addition predominantly helps reducing the hallucination rate for API names and parameters, which improves the overall response rate of NL2DSL generation. This supports our initial premise: adding tool descriptions (like it is done in planning tasks) along with few-shot code samples helps improve reliability of plan generation.

5.3 Regular Function Definition vs Semantic Function Definitions

We used a Fine-Tuned model as baseline for this experiment (Table 4). Based on the insights from the previous step, we used 20 few-shots for TST along with including FDs.

Looking at metrics in columns for % made-up API names and % made-up parameter keys, we see that the hallucination rate is, in general, increasing for RAG based approach. However, we need to keep in mind that a fine-tuned model on the function names is hard to beat as it has been trained on 67,000 samples compared to only 20 few-shots that have been added to the RAG model.

Within the RAG approaches, comparing rows 1 and 2 ("TST + FD" vs "TST + SFD"), SFD results in a slight drop in average similarity and an increase in the unparsed rate and hallucination rate for parameters. This indicates that the approach

Model	Avg. Similarity	%Unparsed flows	%Made-up API names	%Made-up API parameters
Pre-trained + FD	−0.01	+2.29	−2.17	−6.93
TST wo FD	0	**+0.52**	−1.15	+0.52
TST + FD	**+0.02**	+0.83	**−2.7**	**−7.06**

Table 3: Impact of selecting **20 few-shot** samples using TST vs. Pre-trained Model with and without Function Definitions using GPT4 model. The baseline uses Pre-trained Transformer Model without API Function Definitions.

Model	Avg. Similarity	%Unparsed flows	%Made-up API names	%Made-up API parameters
TST + FD	**0**	**−5.3**	+1.7	**+1.11**
TST + SFD	−0.01	−1.43	+1.21	+6.76
TST + FD + SFD	**0**	−2.74	**+0.94**	+2.03

Table 4: Impact of adding API or tool related metadata on performance (with GPT-4 model and 20 few-shots). FD refers to including only metadata for APIs present in few-shots. SFD refers to extracting APIs similar to the input query (refer to Section 3) for details. The baseline uses fine-tuned Codex model.

to simply add semantically similar API metadata for a query is not useful for DSL generation. We get better similarity, and reduced hallucination rate, when we include the API Function Definitions for the samples selected by TST (as shown in Row 1).

5.4 Out of Domain APIs

To compare the impact of RAG on unseen APIs, not available for fine-tuning, we created an out of domain test set. We selected 10 APIs, and discarded the flows containing these APIs from the train set. The test set contains 136 (NL, flow) pairs having these APIs.

We share the results in Table 5. The baseline is a fine-tuned Codex model with the updated training data. The RAG-based approach notably enhances average similarity (by 7 pts) and reduces API hallucinations (by 1.5 pts) for out of domain APIs. This indicates that when samples are not present in the train set, grounding with RAG context can provide the LLM support for improving code quality.

However, fine-tuned model outperforms RAG model in terms of syntactic errors and parameter key hallucinations. The role of few-shots in informing the syntax of the output code cannot be substituted with just adding function definitions. Since, it is hard to obtain the examples for unseen APIs, we need to find alternate ways to improve syntactic errors. We will look into improving this as future work.

6 Conclusion

Based on the presented results, we see that the role of dynamically selected few-shot samples is very important in making RAG useful for syntactically correct generation of DSL as well as improving code similarity (Table 4). This could be due to the fact that few-shot examples have been successfully teaching the correct syntax to the LLM model. The positive role of relevant few-shot samples in improving RAG's syntactic accuracy is further confirmed by the drop seen for out of domain data. In absence of relevant few-shots for unseen APIs, we chose examples with low similarity, directly impacting the syntactic accuracy (Table 5).

Counter intuitively, with the exception of out-of-domain data, this benefit does not transfer to hallucinated API names and their parameter keys where the fine-tuned model holds the advantage (Table 4). Among RAG approaches (Tables 2 and 3), TST + Regular Function Definitions reduced hallucinations the most. Adding Semantic Function Definitions to TST + FD did not confer any advantage for in-domain APIs, but greatly improved code similarity for out-of domain APIs.

Overall, we were able to significantly improve the performance of RAG for DSL generation, with hallucination rate for API names dropping by 6.29 pts. and by 20 pts for parameter keys (Table 2). The performance of RAG is now comparable to that of fine-tuned model (see Avg. Similarity in Table 4), with better performance for unseen APIs. This reduces the need to fine-tune the model frequently for new APIs saving compute and resources.

Test set	Avg. Similarity	%Unparsed flows	%Made-up API names	%Made-up API parameters
OOD test set	**+0.07**	+12.23	**−1.98**	+13.11
Full test set	0	+1.06	**−0.48**	+3.88

Table 5: Performance of RAG based model (TST + FD + SFD) on out of domain (OOD) and full test sets (with GPT-4 model and 20 few-shots). The baseline is fine-tuned Codex model on updated training data.

7 Ethical Considerations

We used instructions in meta prompt to not respond to harmful queries. This is supplemented with a harms classifier on the input prompt. The Fine-tuned model was shown examples where it should not generate an output and consequently learnt not to respond to queries considered harmful.

References

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Qin Cai, Martin Cai, Caio César Teodoro Mendes, Weizhu Chen, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Yen-Chun Chen, Yi-Ling Chen, Parul Chopra, Xiyang Dai, Allie Del Giorno, Gustavo de Rosa, Matthew Dixon, Ronen Eldan, Victor Fragoso, Dan Iter, Mei Gao, Min Gao, Jianfeng Gao, Amit Garg, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Jamie Huynh, Mojan Javaheripi, Xin Jin, Piero Kauffmann, Nikos Karampatziakis, Dongwoo Kim, Mahoud Khademi, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Ce Liu, Mengchen Liu, Weishung Liu, Eric Lin, Zeqi Lin, Chong Luo, Piyush Madan, Matt Mazzola, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Xin Wang, Lijuan Wang, Chunyu Wang, Yu Wang, Rachel Ward, Guanhua Wang, Philipp Witte, Haiping Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Sonali Yadav, Fan Yang, Jianwei Yang, Ziyi Yang, Yifan Yang, Donghan Yu, Lu Yuan, Chengruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *Preprint*, arXiv:2404.14219.

ChatGPT. 2022. Chatgpt.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Code Llama. 2023. Code llama: Open foundation models for code.

Codestral. 2024. Codestral.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *Preprint*, arXiv:2401.08281.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. *Preprint*, arXiv:2211.10435.

GPT-4. 2023. Gpt-4 technical report.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2021. Jigsaw: Large language models meet program synthesis. *Preprint*, arXiv:2112.02969.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large language models are zero-shot reasoners. *Preprint*, arXiv:2205.11916.

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

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, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Yujia Li, David 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 J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, Yun Wang, Linjun Shou, Ming Gong, and Nan Duan. 2023. Taskmatrix.ai: Completing tasks by connecting foundation models with millions of apis. *Preprint*, arXiv:2303.16434.

Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2023. Improving chatgpt prompt for code generation. *Preprint*, arXiv:2305.08360.

LlamaIndex. 2023. Llamaindex.

Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of github copilot's code suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 1–5.

OpenAI Code Interpretor. 2023. Openai code interpretor.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models. *Preprint*, arXiv:2205.12255.

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

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *Preprint*, arXiv:2201.11227.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *Preprint*, arXiv:2303.07839.

Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2021. In-ide code generation from natural language: Promise and challenges. *Preprint*, arXiv:2101.11149.

8 Example Appendix

8.1 Sample with computed Average similarity

Sample showing how flow similarity is computed for two flows Flow A and Flow B.

```
Query = "Post a message in the channel of teams,
when a new form is created in the forms"

Ground Truth = "triggerOutputs =
await shared_microsoftforms.
CreateFormWebhook({});
outputs_shared_teams_PostMessageToConversation
 = shared_teams.PostMessageToConversation(
{ \"poster\": \"User\" });"

prediction: "triggerOutputs =
await shared_microsoftforms.
CreateFormWebhook({});
outputs_Get_my_profile_V2 =
shared_office365users.MyProfile_V2({});
outputs_shared_teams_PostMessage
= shared_teams.PostMessageToConversation(
{\"poster\": \"User\",\"location\":
\"Channel\"});"

API Functions list in ground_truth  =
[shared_microsoftforms.CreateFormWebhook,
shared_teams.PostMessageToConversation]
```

```
API function list in model generation =
[shared_microsoftforms.CreateFormWebhook,
shared_office365users.MyProfile_V2,
shared_teams.PostMessageToConversation]

Similarity Score = 2/3 = 0.666

Since the functions shared_microsoftforms.
CreateFormWebhook and shared_teams.
PostMessageToConversation are found
in the ground truth.
```

8.2 An example of API metdata

We share a sample of API metadata to highlight the details included in the API description provided to the metaprompt.

```
"shared_outlook.SendEmailV2": {
    "FunctionName": "shared_outlook.
                    SendEmailV2",
    "Description": "This operation sends
                     an email message.",
    "IsInTrainingSet": false,
    "DisplayName": "Send an email (V2)",
        "ParametersInfo": [
            {
               "Key": "emailMessage/To",
                "Type": "String",
                "Summary": "To",
                "Format": "email",
                "Description": "Specify
                    email addresses
                  separated by semicolons
               like someone@contoso.com"
            }, . . . .
                        ],
        "ResponseSchema": [],
        "IsTrigger": false
    }
```