API PACK: A MASSIVE MULTI-PROGRAMMING LANGUAGE DATASET FOR API CALL GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce API Pack, a massive multi-programming language dataset containing more than one million instruction-API calls that seeks to improve the API call generation capabilities of large language models. Our evaluation highlights three key findings: First, fine-tuning on API Pack enables open-source models to outperform both GPT-3.5 and GPT-4 in generating code for completely new API calls. We demonstrate this by fine-tuning CodeLlama-13B on 20,000 Python instances from API Pack. Second, we demonstrate that fine-tuning on a large dataset in one programming language, combined with smaller datasets from others, improves API generation accuracy across multiple languages. Third, we demonstrate the benefits and the need of larger datasets for API generalization, as our experiments show that increasing fine-tuning data to 1 million instances improves generalization to new APIs. To support further research, we open-source the API Pack dataset, trained model, and code at `https://github.com/anonymous/API-Pack`.

1 INTRODUCTION

Large language models (LLMs) have shown promise in assisting with software engineering tasks (Li et al., 2023a; Ozkaya, 2023; Fan et al., 2023; Belzner et al., 2023; Ross et al., 2023; Hou et al., 2023; Ebert & Louridas, 2023; Chang et al., 2024; Wang et al., 2024), with a primary focus on code generation (Chen et al., 2021b; Xu et al., 2022; Sarsa et al., 2022; Vaithilingam et al., 2022; Poesia et al., 2022; Wang et al., 2023b; Shrivastava et al., 2023a; Liang et al., 2023; Zan et al., 2023; Liu et al., 2023a; Shrivastava et al., 2023b; Wei et al., 2023; Muennighoff et al., 2023; Li et al., 2023c; Huang et al., 2023; Lozhkov et al., 2024; Thakur et al., 2024; Mishra et al., 2024). While these advances are significant, developers still face a time-consuming challenge: finding and adapting API code examples from lengthy documentation (Meng et al., 2018). This manual process of searching for appropriate code examples remains cumbersome and inefficient. Although prior research has addressed API intent detection (finding the right API for a task), the more complex challenge of generating specific API call code has received limited attention, particularly given the varying syntax of HTTP protocol elements across programming languages.

To achieve this goal, we create API Pack, a dataset designed to improve the API call generation capabilities of LLMs. With over 1 million instances spanning 10 programming languages, API Pack represents the largest open-source instruction dataset for API call generation and intent detection (see Table 1). Our dataset distinguishes itself from prior works (Zan et al., 2022; Zhang et al., 2023a; Xu et al., 2023b; Patil et al., 2023; Qin et al., 2023) in two key aspects: programming language diversity and dataset size. By including API calls across multiple languages, API Pack enables the first comprehensive study of cross-language skill transfer in API generation, i.e., examining how improvements in one language generalize to others. Additionally, its extensive collection of real-world API examples allows for the evaluation of generalization capabilities through controlled variations in training data volume.

We summarize three key findings from our experiments:

1. Fine-tuning CodeLlama-13B on 20k Python API Pack instances outperforms GPT-3.5 by 10% and GPT-4 by 5% on unseen API calls.
2. Cross-language API call generation can be enabled by a large amount of data in one programming language plus small amounts of data in others.

Table 1: A comparison of API Pack with other instruction datasets for API intent detection and/or API call code generation. The upper section of the table reports the features that each dataset covers, and the bottom section reports the data statistics available (/ means unavailable).

Feature	API Pack (this work)	APIBench (Gorilla)	ToolBench	ToolBench (ToolLLM)	API Bank	ToolAlpaca	ToolFormer
API call intent detection?	✓	✓	✓	✓	✓	✓	✓
API call code generation?	✓	✓	✓	✗	✗	✗	✗
Multi-programming language?	✓(10)	✗(Python)	✓(Curl, Python)	✗	✗	✗	✗
Multi-API call scenario?	✗	✗	✓	✓	✓	✓	✓
Data generation method	custom	self-instruct	self-instruct	custom	custom	custom	custom
# of Sources	4	3	8	1	53	/	5
# of APIs / Tools	11,213	1,645	8	16,464	53	400	5
# of API calls	1,128,599	16,450	/	37,204	568	3,938	9,400
# of Instances	1,128,599	16,450	2,746	12,657	264	3,938	22,453

3. Increasing fine-tuning data from 0 to 1 million instances improves generalization to new APIs, demonstrating clear benefits of larger datasets for API generalization.

This paper is structured as follows: We begin with a review of related work (Section 2), followed by a detailed description of the API Pack dataset construction (Section 3). Our experimental method for model fine-tuning and evaluation is presented in Section 4, with key findings discussed in Section 5. We conclude in Section 6 with a discussion of limitations and future directions. Additional considerations regarding ethics and broader impacts are addressed in Appendices A.1 and A.2, respectively. The API Pack dataset is made available under the Creative Commons Attribution 4.0 International License, with accompanying source code released under the MIT License.

2 Related Work

2.1 Methods to Generate Instruction Data with LLMs

The manual creation of instruction datasets is a time-consuming and resource-intensive process (Xu et al., 2023a). To address this challenge, researchers have developed automated methods leveraging LLMs. Two notable approaches in this domain are Self-Instruct (Wang et al., 2023a) and Evol-Instruct (Xu et al., 2023a), both of which focus on generating and filtering instruction-response pairs. Self-Instruct utilizes in-context learning to modify seed examples and employs ROUGE-L similarity and heuristics for filtering. In contrast, Evol-Instruct uses targeted prompting for data generation and relies on predefined heuristics for filtering.

For code-specific datasets, hybrid LLM-based pipelines offer an alternative approach. These pipelines typically combine high-quality code samples from human with LLM-generated instructions (Patil et al., 2023; Xu et al., 2023b). While human verification remains the gold standard Wang et al. (2023a), it becomes impractical at scale. Recent work has instead adopted powerful LLMs like GPT-4 and Mistral as automated judges, using example-guided heuristics derived from human-reviewed data (Liu et al., 2023c). Similar techniques have been applied to evaluate instruction complexity (Chen et al., 2023; Lu et al., 2023).

2.2 LLMs for API Call Code Generation and Intent Detection

LLMs for code generation have predominantly focused on general coding tasks, as evaluated by benchmarks like MBPP (Austin et al., 2021), HumanEval (Chen et al., 2021a), and their variants (Liu et al., 2023b; Zheng et al., 2023; Peng et al., 2024). Recent work has expanded into two specific API-related domains: API call intent detection and API call code generation.

API call intent detection focuses on matching natural language tasks to appropriate API endpoints. LLMs designed for this purpose (Zan et al., 2022; Zhang et al., 2023a; Qin et al., 2023; Li et al., 2023b; Tang et al., 2023; Yang et al., 2023; Schick et al., 2023) often operate in hybrid architectures, where the LLM identifies the relevant API endpoint(s) while other components handle the code generation. These studies have explored both single-API and multi-API intent detection. Achieving a strong performance in the latter remains a challenge Qin et al. (2023). At present, this is likely the

way most function calling LLMs have been trained to operate (Srinivasan et al., 2023; Yuan et al., 2024; Chen et al., 2024).

Our work, however, focuses on training LLMs for API call code generation, which remains under explored compared to API call intent detection. To the best of our knowledge, only two projects have previously worked on this challenge: Gorilla (Patil et al., 2023), which generates API calls to load pre-trained machine learning models from known model hubs, and ToolBench (Xu et al., 2023b), which focuses on improving LLM's tool manipulation capabilities for a few applications.

3 API PACK

API Pack is an instruction dataset with over one million instances. Each instance contains an input-output pair plus additional information about the API and respective endpoints. The inputs are instructions for finding an API call to solve a coding task, including a task description in software engineering language and the name of the API. The outputs are API call examples, specifically HTTP request code snippets curated from OpenAPI specification (OAS) files. The data in API Pack is sourced from four hubs that store OAS files: RapidAPI[1], APIGurus[2], Swaggerhub[3], and IBM's public API Hub[4]. To the best of our knowledge, these API Hubs were the only sources with publicly available OAS files at a large scale at the time we collected data. From these sources, we kept all the APIs that passed our filtering criteria (see Section 3.1). Table 2 summarizes the total number of APIs, unique endpoints, and total instances that are part of API Pack.

Table 2: Final count of data curated per source, where each instance contains one API call

Source	APIs	Unique Endpoints	Total Instances
RapidAPI	4,115	21,525	270,095
APIs Gurus	1,980	37,097	495,533
SwaggerHub	5,045	26,747	345,765
IBM API Hub	73	2,884	17,206
Total	**11,213**	**88,253**	**1,128,599**

The construction of API Pack involves four main stages: data pre-processing (Section 3.1), API Database (DB) creation (Section 3.2), instruction generation (Section 3.3), and data validation (Section 3.4). Figure 1 illustrates the overall pipeline.

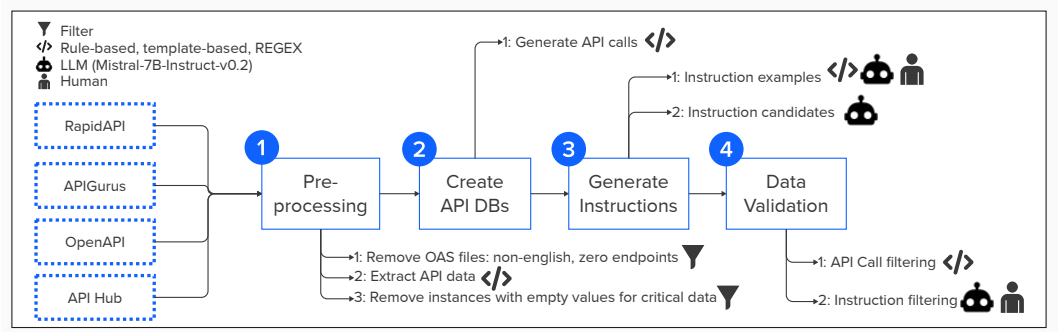

Figure 1: Dataset curation pipeline.

[1]https://rapidapi.com/categories
[2]https://apis.guru/
[3]https://app.swaggerhub.com/search
[4]https://developer.ibm.com/apis/

3.1 DATA PRE-PROCESSING

This process involves two main steps. First, we filter out OAS files with non-English metadata (less than 1%) and those with zero endpoints, as they cannot be used to generate API calls. Next, we extract key information from the remaining OAS files. At the API level, we collect the name, description, and provider. At the endpoint level, we extract the name, functionality, description, method, and path. To ensure data quality, we remove instances missing critical information needed for generating API calls (e.g., method, path, or endpoint name) or instructions (e.g., functionality or description). This step ensures that all dataset instances include the necessary details for generating accurate API call examples and instructions.

3.2 CREATE API DATABASES

We construct an API database (DB) from the pre-processed data, where each instance contains endpoint and API details. Using the OpenAPI Snippet[5] library, we generate API calls in 10 programming languages (cURL, libcurl, Java, Node.js, Python, Go, Ruby, PHP, Swift, and JavaScript) for each instance. This process is applied to data from RapidAPI, APIGurus, and OpenAPI. For IBM API Hub, we directly extract API calls from OAS files, as these already include calls in multiple languages. The selection of programming languages is based on two factors: (1) the 10 languages align with those highlighted in the StarCoder report (Li et al., 2023c), and (2) the OpenAPI Snippet library's support for these languages. Appendix A.7 provides details on the API DB structure and language diversity across sources.

3.3 INSTRUCTIONS GENERATION

This process involves creating high-quality instruction examples and generating instruction candidates based on them. First, we randomly select three endpoints from each API DB and use their details (e.g., functionality, description, endpoint name, path) along with the API name to fill predefined instruction templates. This results in three instruction examples per API DB file. For the API Gurus and IBM API Hub sources, three co-authors of this work manually refine the generated instruction examples. The refinement criteria are: 1) correcting grammatical errors, 2) removing unnecessary information, and 3) ensuring the API name is included and correct in the examples. Most annotations were done independently, but annotators discussed non-standard cases to reach consensus. To streamline the refinement process for Swaggerhub and RapidAPI sources, we use an LLM (Mistral-7B-Instruct-v0.2[6]) instead of human effort. We create a prompt (see Prompt 1 in Appendix A.10) based on manual review heuristics and use it to task the LLM with refining instruction examples for these sources. Next, we use the high-quality instruction examples to generate five instruction candidates for each API DB instance. We prompt the LLM with the endpoint's information and provide the high-quality instruction examples as in-context examples (see Prompt 2 in Appendix A.10). Figure 7 in Appendix A.8 shows an instance with one instruction candidate and its corresponding API call. Appendix A.9 presents all five candidates generated for the same instance.

3.4 DATA VALIDATION

The data validation process involves three main steps: 1) verifying the validity of API calls as HTTP request examples in a given programming language, 2) assessing the quality of generated instructions, and 3) selecting the highest-quality instruction for fine-tuning.

First, to verify API calls, we ensure that the `endpoint_name` and HTTP method (e.g., GET, POST) in the API call match the instance data. We use regular expressions to validate the URL format, accounting for placeholders in the domain, path parameters, or query parameters. Additionally, we check that the programming language keywords in the API call correspond to the language ID assigned to each instance.

Second, to assess instruction quality, we randomly sampled 121 instances, each with one API call and five instructions, totaling 605 instructions. A co-author with expertise in software development manually classified these instructions as good or bad, with a second annotator providing feedback.

[5]https://www.npmjs.com/package/openapi-snippet

[6]https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2

Bad instructions typically: 1) contain multiple instructions instead of one, 2) include unnecessary text, or 3) incorrectly use the API name. Based on these traits, we created three prompts with fixed in-context examples to automatically annotate instructions using an LLM. We used the LLM (Mistral-7B-Instruct-v0.2) to annotate the 605 instructions and compared the results with human annotations. We selected the prompt with the best performance (see Prompt 4 in Appendix A.10) and used it to classify all instructions in our dataset. We then removed instances with fewer than two good instructions.

Third, to select the best instruction candidate from the good ones, we calculated the likelihood that an LLM would regenerate the input text used to create the instruction. We prompted Mistral-7B-Instruct-v0.2 with each instruction candidate (see Prompt 3 in Appendix A.10) and obtained the log probability of each token for the regenerated input text. We computed the mean log probability (`input_tokens_mean`) and selected the instruction annotated as good with the highest `input_tokens_mean` for fine-tuning. Our final dataset includes 1,128,599 instances, each with a valid API call and at least two high-quality instructions. Appendix A.3 details the data instances filtered out at each pipeline stage.

4 Experiments and Evaluation Framework

To optimize the instruction-following capabilities of the language models, we post-process API Pack into two instruction-tuning templates. The **0-shot-tuning** template targets scenarios where the output is expected to be a straightforward inference from the given input. The **3-shot-tuning** template, with random selections within the selected API, emphasizes the model's ability to learn and generate output with in-context learning. The **3-shot-tuning** template is available in Appendix 5. Model generation parameters follow the BigCodeBench (Zhuo et al., 2024).

4.1 Experimental Settings

A. Selecting a base model: Our first experimental study serves the purpose of selecting a base model for the rest of our experiments. We fine-tune Mistral 7b (Jiang et al., 2023), CodeLlama 7b and 13b, as well as Llama 2 13b (Touvron et al., 2023) on a subset of API Pack (20,000 instances in Python programming language). We evaluate the performance of resulting models and select the best-performing base model.

B. Inference with retrieval: Our second experiment aims to understand the influence of retrieval augmentation on model generalization. We evaluate the models under five distinct prompt settings during test time:

- **0-shot**: No API examples provided for the model.
- **3-shot random**: Three randomly selected API examples.
- **3-shot retrieved**: Three retrieved relevant API examples.
- **3-shot retrieved & re-ranked**: Five retrieved API examples, with three selected using a re-ranker model in the final prompt.
- **3-shot oracle**: Three retrieved relevant API examples. Replacing one of the randomly selected retrieved API examples with the oracle example.

The **3-shot random inference** uses the same prompt template as **3-shot-tuning**. We use *bge-large-en-v1.5* (Zhang et al., 2023b) as the embedding model for retrieval and *bge-reranker-large* (Xiao et al., 2023) for re-ranking the API examples. Appendix A.6 includes more details of the inference pipeline used to evaluate the models' performance.

C. Cross-language generalization: To test the model's ability to generalize to new programming languages, we supplement a cURL dataset of 100,000 instances with 1,000 instances from each of the 9 additional languages in API Pack: Go, Java, JavaScript, libcurl, Node.js, PHP, Python, Ruby, and Swift. The goal is to determine if a model can generalize to new languages without requiring a large amount of multi-programming language data. Notably, these instances across different languages invoke the same API functionality, allowing us to evaluate how well models can translate the same API invocation across different programming languages.

D. Scaling experiment: We conduct a scaling experiment to investigate whether more API data improves a model's generalization ability for unseen APIs. We fine-tune models on progressively larger API datasets with unique API calls on 0k, 10k, 20k, 40k, 80k, 100k, and 1 million instances. Our hypothesis is that exposure to a greater scale and diversity of APIs during fine-tuning will improve the model's ability to generalize to new, unseen APIs.

4.2 EVALUATIONS

To measure the generalization capabilities enabled by API Pack, we establish a comprehensive evaluation framework spanning three levels of complexity for API call generation:

- **Level 1**: Seen APIs and endpoints. This level assesses generalization to new instructions.
- **Level 2**: Seen APIs and new endpoints. This level tests generalization to new endpoints of known APIs.
- **Level 3**: Unseen APIs and endpoints. This level validates performance on entirely new APIs.

The process of data splitting is detailed in Appendix A.4, and training hyperparameters are provided in Appendix A.5. To evaluate the accuracy of the generated Endpoints and API calls, we use the `SequenceMatcher` class from Python's `difflib` library. This algorithm computes a similarity ratio between two sequences by finding the longest contiguous matching subsequences while considering certain elements as insignificant. The similarity ratio r is calculated as:

$$r = \frac{2M}{T},$$

where M is the total number of matching characters (excluding insignificant elements), and T is the total number of characters in both sequences (Black, 2021). Insignificant elements refer to parts of the API requests that do not affect functionality, such as whitespace, formatting differences (e.g., spaces, tabs, newlines), and variable naming conventions. A generated output is considered correct if the similarity ratio meets or exceeds a heuristic threshold of 0.9.

This evaluation focuses on structural and semantic correctness rather than exact syntactic matches. While execution-based metrics like pass rate (Chen et al., 2021b) or hallucination rate (Jain et al., 2024) could offer more robust evaluation, our dataset prioritizes privacy by removing personally identifiable information (PII) from the API database, making execution-based evaluation challenging. We discuss these evaluation limitations and potential future directions in Section 6.

5 EXPERIMENTAL RESULTS

5.1 FINE-TUNED CODELLAMA EXCELS IN API CALL GENERATION

We conducted a comprehensive evaluation comparing models fine-tuned on 20,000 Python instances from API Pack against GPT-3.5 and GPT-4. The evaluation assessed fine-tuned open-source models across all three difficulty levels, while GPT-3.5 and GPT-4 were evaluated only on Level 3 due to fine-tuning cost constraints. Table 3 presents these results, demonstrating that open-source LLMs fine-tuned on API Pack can match or exceed the performance of leading proprietary models in generating unseen API calls.

Our analysis reveals three key findings:

1. **Advantage of CodeLlama-13b:** The fine-tuned CodeLlama-13b model consistently outperforms other models, achieving the highest API call accuracy across all evaluation levels, particularly in the 3-shot retrieved setting. For Level 3, it achieves 49.5% accuracy, compared to 42.5% for Mistral-7b and 44.2% for Llama-2-13b.
2. **3-shot vs. 0-shot Training:** Models fine-tuned with the 3-shot-tuning template consistently outperform those trained with the 0-shot-tuning template. For example, CodeLlama-13b's Level 3 accuracy improves from 44.1% (0-shot training) to 49.5% (3-shot training) when using 3-shot retrieved prompts.

3. **Outperforming GPT Models:** Our fine-tuned CodeLlama-13b model surpasses both GPT-3.5 and GPT-4 on Level 3 evaluations. In the 3-shot retrieved setting, CodeLlama-13b achieves 49.5% API call accuracy, compared to 39.5% for GPT-3.5 and 44.3% for GPT-4. This 5.2 percentage point improvement over GPT-4 demonstrates the effectiveness of our fine-tuning approach with API Pack.

Table 3: Evaluation for models fine-tuned with 20k Python API dataset, including a comparison with (not-fine-tuned) CodeLlama-13b, GPT-3.5, and GPT-4. Levels 1 and 2 are not reported for GPT-3.5 and GPT-4 due to proprietary fine-tuning processes. Note that 3-shot (retre) is short for 3-shot retrieved.

Model	Fine-tuning template	Testing	Evaluation Accuracy (%)					
			Level 1		Level 2		Level 3	
			Intent	API Call	Intent	API Call	Intent	API Call
Mistral-7b	0-shot	0-shot	17.2	10.9	14.1	11.4	14.3	11.2
		3-shot (retre)	42.0	29.7	35.4	28.7	39.1	29.1
	3-shot	0-shot	40.5	28.5	24.0	18.3	15.2	12.1
		3-shot (retre)	**64.1**	55.4	49.1	42.8	50.8	42.5
CodeLlama-7b	0-shot	0-shot	8.1	6.1	10.0	7.0	11.0	7.8
		3-shot (retre)	52.6	42.6	43.6	35.9	50.2	40.1
	3-shot	0-shot	12.1	9.3	13.7	10.2	16.8	13.0
		3-shot (retre)	60.6	52.7	54.1	47.3	55.9	49.1
Llama-2-13b	0-shot	0-shot	9.4	6.2	11.6	9.0	10.9	8.4
		3-shot (retre)	44.5	33.9	45.4	35.6	46.7	39.1
	3-shot	0-shot	15.7	10.2	14.0	11.2	11.7	9.6
		3-shot (retre)	59.5	51.5	50.8	44.3	52.7	44.2
CodeLlama-13b	0-shot	0-shot	9.8	6.8	10.8	8.1	12.1	8.5
		3-shot (retre)	55.6	44.4	50.6	43.3	52.3	44.1
	3-shot	0-shot	14.4	10.3	15.9	13.3	14.2	8.9
		3-shot (retre)	63.5	**55.5**	**56.8**	**51.4**	**56.1**	**49.5**
	none	0-shot	0.1	0.0	0.2	0.0	0.1	0.0
		3-shot (retre)	49.2	46.2	36.3	34.4	40.7	38.5
gpt-3.5-1106	none	0-shot	-	-	-	-	1.0	0.7
	none	3-shot (retre)	-	-	-	-	47.2	39.5
gpt-4-1106	none	0-shot	-	-	-	-	0.2	0.1
	none	3-shot (retre)	-	-	-	-	53.5	44.3

5.2 Retrieval Augmentation Improves API Call Generation

Table 4: Performance comparison for different retrieval methods in 3-shot prompting. Note that 3-shot (retre & rerank) is short for 3-shot retrieved & re-ranked.

Model	Testing	Evaluation Accuracy (%)					
		Level 1		Level 2		Level 3	
		Endpoint	API Call	Endpoint	API Call	Endpoint	API Call
Mistral-7b	3-shot (rand)	54.5	41.8	48.2	41.2	45.2	37.0
	3-shot (retre)	64.1	55.4	49.1	42.8	50.8	42.5
	3-shot (retre & rerank)	63.0	53.6	49.0	42.2	51.5	43.9
	3-shot (oracle)	75.8	68.4	62.5	56.8	60.2	53.1
CodeLlama-13b	3-shot (rand)	49.2	38.6	49.8	43.6	50.0	41.4
	3-shot (retre)	63.5	55.5	56.8	51.4	56.1	49.5
	3-shot (retre & rerank)	61.0	52.9	55.1	49.2	55.9	49.3
	3-shot (oracle)	78.5	71.2	69.8	64.1	65.9	59.3

Table 4 shows how different retrieval methods affect 3-shot API call generation. Our analysis highlights four key findings:

1. **Advantages of Retrieved Examples:** 3-shot (retre) consistently outperforms 3-shot (rand) for both Mistral-7b and CodeLlama-13b models across all evaluation levels.

2. **Limitations of Generic Re-ranking:** Interestingly, 3-shot (retre & rerank) does not consistently improve upon 3-shot (retre) and sometimes performs slightly worse (e.g., Level 1 drops from 63.5% to 61.0% for CodeLlama-13b). Our 3-shot (oracle) results show that better example selection can improve performance. However, the current re-ranker fails to capture these gains, suggesting that a fine-tuned re-ranker adapted to API characteristics might be more effective.
3. **Performance Gap Across Levels:** We observe a notable performance gap between retrieval methods for Level 1, which narrows for Levels 2 and 3. This trend suggests that retrieval is particularly effective for seen APIs and endpoints (Level 1).
4. **Endpoint vs. API Call Accuracy:** Across all conditions, Endpoint accuracy is consistently higher than API Call accuracy. This indicates that identifying the correct endpoint is easier than generating the full API call, which requires accurately specifying more parameters, their types, and any required formatting.

These results demonstrate that at inference time, 3-shot (retre) is the most effective approach for fine-tuned Mistral-7b and CodeLlama-13b models compared to 3-shot (rand) and 3-shot (retre & rerank).

5.3 Cross-Language Generalizations in API Call Performance

Figure 2 compares three fine-tuning approaches for cross-language API call performance: a model fine-tuned exclusively on 100,000 instances of 'cURL' data, or cURL Model, three Expert Models, each fine-tuned on 100,000 samples of 'cURL', 'Python', and 'Java' data separately, and a Mixture Model fine-tuned on 100,000 instances of 'cURL' data with additional samples of 1,000 instances each for nine different languages. Since all these instances invoke the same API functionality across different languages, this comparison directly evaluates the models' ability to translate equivalent API calls across programming languages.

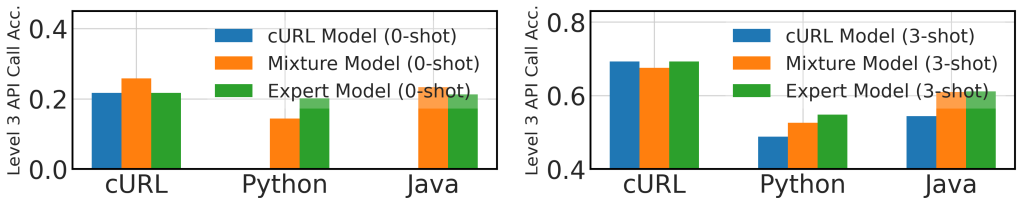

Figure 2: Comparison of 0-shot and 3-shot API call performance for different models in cURL, Python, and Java. Note that the Expert Models are specific to each programming language.

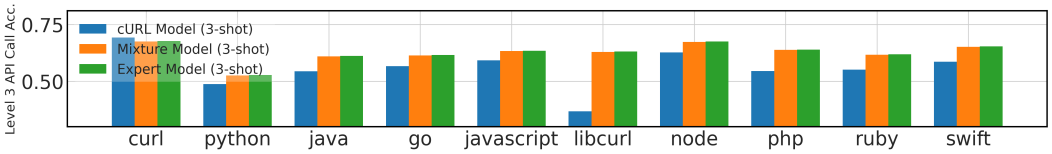

Figure 3: Three-shot performance in ten languages across different models.

The comparsion shows that without fine-tuning, models struggle to generalize to new programming languages in 0-shot scenarios. However, in 3-shot settings, performance improves significantly, demonstrating the benefits of in-context learning.

Figure 3 illustrates that the Mixture Model performs comparably to Expert Models across ten programming languages in 3-shot testing. This demonstrates the Mixture Model's ability to generalize effectively without significant performance loss, highlighting its robustness and versatility for multi-language programming applications.

Overall, these findings indicate that cross-language API call generation is achievable by fine-tuning models with a large dataset in one language and smaller datasets in others, eliminating the need for extensive data in each target programming language.

5.4 SCALING INSTRUCTION DATASET HELPS GENERALIZATION

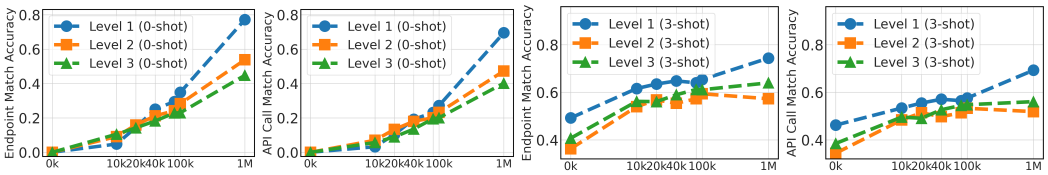

Figure 4: Scaling instruction dataset, with 3-shot fine-tuning template, on CodeLlama-13b with 0-shot and 3-shot retrieval evaluations. The x-axis represents the log scale size of fine-tuning data from API Pack. The y-axis is Endpoint or API call accuracy.

Figure 4 shows that 0-shot performance increases with the fine-tuning dataset size (ranging from 0k for the non-finetuned baseline to 1 million instances from API Pack using a 3-shot fine-tuning template). We observe a roughly linear rise in both endpoint match accuracy and API call match accuracy as the dataset grows. This trend is consistent across all evaluation levels (1, 2, and 3), indicating that larger datasets enhance the model's ability to generate API calls without examples, likely due to exposure to more diverse API patterns and structures. For 3-shot prompting, the graph also shows performance gains as the dataset size increases, though these gains are smaller than in the 0-shot case, especially for level 3 cases.

For Level 1 (seen APIs and endpoints), there is a similar linear improvement as in the 0-shot case, but with a gentler slope. This suggests that even with in-context examples, larger fine-tuning datasets further improve performance on familiar APIs. However, for Levels 2 (new endpoints of known APIs) and 3 (unseen APIs), performance initially improves but then plateaus as fine-tuning data approaches 1 million instances. The plateau is most evident for Level 3, our most challenging evaluation scenario. These findings indicate that there may be diminishing returns from scaling instruction data, particularly for the most difficult generalization tasks (Level 3).

5.5 ADDITIONAL EXPERIMENTS

We conducted two experiments to assess the benefits of API Pack: First, to evaluate the impact of using multiple API sources, we fine-tuned CodeLlama-13b on 20k instances from ToolBench, which only includes APIs from RapidAPI. Second, to test our data filtering pipeline, we created a Non-filtered API Pack by replacing 45% of the filtered instructions by the LLM with non-filtered ones, and fine-tuned CodeLlama-13b on 20k instances.

Table 5: Performance comparison for ToolBench, Non-filtered API Pack, and API Pack in different shot settings.

Training Dataset	Testing	Evaluation Accuracy (%)					
		Level 1		Level 2		Level 3	
		Endpoint	API Call	Endpoint	API Call	Endpoint	API Call
ToolBench	0-shot	5.7	5.7	8.1	8.0	7.3	7.0
	3-shot	44.5	37.8	40.7	36.4	43.7	38.1
Non-filtered API Pack	0-shot	10.3	8.3	12.8	10.3	12.2	8.4
	3-shot	62.4	54.9	54.9	48.7	55.7	49.5
API Pack	0-shot	14.4	10.3	15.9	13.3	14.2	8.9
	3-shot	**63.5**	**55.5**	**56.8**	**51.4**	**56.1**	49.1

Table 5 presents the results of these experiments alongside our API Pack performance. Key findings from these experiments include:

1. **Multi-source Advantage**: Training with APIs from a single source (ToolBench) reduces performance in both 0-shot and 3-shot settings compared to API Pack, which aggregates data from four sources. This demonstrates the benefit of our multi-source approach.

2. **Data-filtering Importance**: Introducing non-filtered instructions (Non-filtered API Pack) leads to a slight performance drop compared to the fully filtered API Pack. This indicates that while our filtering process contributes to dataset quality, its overall impact on performance appears limited, suggesting room for simplification or refinement in future iterations.

We also evaluated the performance of mixing a subset of 50,000-entries of API Pack with Magicoder dataset and fine-tuned CodeLlama-13b model. The resulting model shows an increase of over 35.3% in API call code generation accuracy for Level 3, specifically with the 3-shot setting. This improvement does not come at the expense of general coding efficiency, as the resulting model still performs well on benchmarks such as HumanEval+ and MBPP. See Table 6 for further details.

Table 6: Evaluation for Code Generation with CodeLlama-13b

Data Mixture	Bench. (pass@10)		Level 3 (3-shot)
	HumanEval+	MBPP	Endpoint
-	47.8	58.3	-
Magicoder	60.8	66.4	17.0
Magicoder + API Pack	61.3	64.3	52.3

6 Conclusion and Future Work

We introduced API Pack, a multi-programming language dataset containing over 1 million instruction-API call pairs to transform software development workflows by enabling LLMs to accurately generate API calls from natural language instructions. Our experimental results demonstrated three key achievements: First, fine-tuning CodeLlama-13B on 20,000 Python instances from API Pack enables it to outperform GPT-3.5 and GPT-4 in generating unseen API calls. Second, we established that model performance continues to improve as training data scales to 1 million instances, assisting generalization to unseen APIs. Third, we showed that cross-language API call generation can be achieved by combining extensive training data in one programming language with smaller amounts from other languages.

We acknowledge that the current API Pack exhibits several limitations. Our evaluation method, while systematic, could benefit from execution-based metrics like pass rates or hallucination rates, but our focus on privacy protection and PII removal currently makes such testing difficult. Additionally, API Pack was not designed for multi-API call scenarios, and its requirement for explicit API name specification may limit application in complex software development scenarios. Programming language coverage is constrained by code generation library limitations, potentially excluding languages of interest to end-users. Furthermore, while API-gurus operates under a CC0-1.0 license, the limited availability of API-level license information may restrict use in proprietary development contexts.

Looking ahead, we plan to address these challenges through several initiatives. We will explore privacy-preserving execution-based evaluation methods using synthetic endpoints or sanitized environments. Our roadmap includes expanding the dataset to support multi-API scenarios and task-based instructions, while also increasing programming language diversity. Finally, we aim to improve API-level license documentation to better serve the needs of all potential users.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models, August 2021. URL http://arxiv.org/abs/2108.07732. arXiv:2108.07732 [cs].

Lenz Belzner, Thomas Gabor, and Martin Wirsing. Large language model assisted software engineering: prospects, challenges, and a case study. In *International Conference on Bridging the Gap between AI and Reality*, pp. 355–374. Springer, 2023.

Paul E. Black. Ratcliff/obershelp pattern recognition. https://www.nist.gov/dads/HTML/ratcliffObershelp.html, January 8 2021. In Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. (Accessed DATE).

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45, 2024.

Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, and Hongxia Jin. AlpaGasus: Training A Better Alpaca with Fewer Data, November 2023. URL http://arxiv.org/abs/2307.08701. arXiv:2307.08701 [cs].

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. Evaluating Large Language Models Trained on Code, July 2021a. URL http://arxiv.org/abs/2107.03374. arXiv:2107.03374 [cs].

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.

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

Christof Ebert and Panos Louridas. Generative AI for Software Practitioners. *IEEE Software*, 40(4):30–38, July 2023. ISSN 1937-4194. doi: 10.1109/MS.2023.3265877. URL https://ieeexplore.ieee.org/abstract/document/10176168?casa_token=JPr1zeiL9IYAAAAA:noU0xEJ-kpRsoaLTTNiSFttcQ_Fw1lBtZRQWnykXIz6rRNvIW4qW-5nljQZYP7H_dH1yJ-3qc-bW. Conference Name: IEEE Software.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*, 2023.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review, September 2023. URL http://arxiv.org/abs/2308.10620. arXiv:2308.10620 [cs].

Qing Huang, Zhenyu Wan, Zhenchang Xing, Changjing Wang, Jieshan Chen, Xiwei Xu, and Qinghua Lu. Let's chat to find the apis: Connecting human, llm and knowledge graph through ai chain. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 471–483. IEEE, 2023.

Nihal Jain, Robert Kwiatkowski, Baishakhi Ray, Murali Krishna Ramanathan, and Varun Kumar. On mitigating code llm hallucinations with api documentation, 2024. URL https://arxiv.org/abs/2407.09726.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Llio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothe Lacroix, and William El Sayed. Mistral 7B, October 2023. URL http://arxiv.org/abs/2310.06825. arXiv:2310.06825 [cs].

Hongxin Li, Jingran Su, Yuntao Chen, Qing Li, and ZHAO-XIANG ZHANG. Sheetcopilot: Bringing software productivity to the next level through large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 4952–4984. Curran Associates, Inc., 2023a. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/0ff30c4bf31db0119a6219e0d250e037-Paper-Conference.pdf.

Minghao Li, Feifan Song, Bowen Yu, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A Benchmark for Tool-Augmented LLMs, April 2023b. URL http://arxiv.org/abs/2304.08244. arXiv:2304.08244 [cs].

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, Joo 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 Muoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: may the source be with you!, December 2023c. URL http://arxiv.org/abs/2305.06161. arXiv:2305.06161 [cs].

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as Policies: Language Model Programs for Embodied Control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500, May 2023. doi: 10.1109/ICRA48891.2023.10160591. URL https://ieeexplore.ieee.org/abstract/document/10160591?casa_token=NZCPW7T2O5QAAAAA:lnnQxWsEhgimKw52mjcQJ-GMER2nOCA11yJHSUvZGA_VZiHcM_qYfKBnd2GCRDbNcLGakL2SgQ.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 21558–21572. Curran Associates, Inc., 2023a. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation, October 2023b. URL http://arxiv.org/abs/2305.01210. arXiv:2305.01210 [cs].

Wei Liu, Weihao Zeng, Keqing He, Yong Jiang, and Junxian He. What Makes Good Data for Alignment? A Comprehensive Study of Automatic Data Selection in Instruction Tuning, December 2023c. URL http://arxiv.org/abs/2312.15685. arXiv:2312.15685 [cs].

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Keming Lu, Hongyi Yuan, Zheng Yuan, Runji Lin, Junyang Lin, Chuanqi Tan, Chang Zhou, and Jingren Zhou. #InsTag: Instruction Tagging for Analyzing Supervised Fine-tuning of Large Language Models, August 2023. URL http://arxiv.org/abs/2308.07074. arXiv:2308.07074 [cs].

Michael Meng, Stephanie Steinhardt, and Andreas Schubert. Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication*, 48(3):295–330, July 2018. ISSN 0047-2816. doi: 10.1177/0047281617721853. URL https://doi.org/10.1177/0047281617721853. Publisher: SAGE Publications Inc.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, et al. Granite code models: A family of open foundation models for code intelligence. *arXiv preprint arXiv:2405.04324*, 2024.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. OctoPack: Instruction Tuning Code Large Language Models, August 2023. URL http://arxiv.org/abs/2308.07124. arXiv:2308.07124 [cs].

Ipek Ozkaya. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software*, 40(3):4–8, 2023.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large Language Model Connected with Massive APIs, May 2023. URL http://arxiv.org/abs/2305.15334. arXiv:2305.15334 [cs].

Qiwei Peng, Yekun Chai, and Xuhong Li. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. *arXiv preprint arXiv:2402.16694*, 2024.

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

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs, July 2023. URL http://arxiv.org/abs/2307.16789. arXiv:2307.16789 [cs].

Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. The programmers assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*, pp. 491–514, 2023.

Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, pp. 27–43, 2022.

Timo Schick, Jane Dwivedi-Yu, Roberto Dess, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools, February 2023. URL http://arxiv.org/abs/2302.04761. arXiv:2302.04761 [cs].

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pp. 31693–31715. PMLR, 2023a.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-Level Prompt Generation for Large Language Models of Code. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 31693–31715. PMLR, July 2023b. URL https://proceedings.mlr.press/v202/shrivastava23a.html. ISSN: 2640-3498.

Venkat Krishna Srinivasan, Zhen Dong, Banghua Zhu, Brian Yu, Damon Mosk-Aoyama, Kurt Keutzer, Jiantao Jiao, and Jian Zhang. Nexusraven: a commercially-permissive language model for function calling. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. ToolAlpaca: Generalized Tool Learning for Language Models with 3000 Simulated Cases, September 2023. URL http://arxiv.org/abs/2306.05301. arXiv:2306.05301 [cs].

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3):1–31, 2024.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models, July 2023. URL http://arxiv.org/abs/2307.09288. arXiv:2307.09288 [cs].

Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pp. 1–7, 2022.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 2024.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-Instruct: Aligning Language Models with Self-Generated Instructions, May 2023a. URL http://arxiv.org/abs/2212.10560. arXiv:2212.10560 [cs].

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. CodeT5+: Open Code Large Language Models for Code Understanding and Generation, May 2023b. URL http://arxiv.org/abs/2305.07922. arXiv:2305.07922 [cs].

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source Code Is All You Need, December 2023. URL http://arxiv.org/abs/2312.02120. arXiv:2312.02120 [cs].

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. C-Pack: Packaged Resources To Advance General Chinese Embedding, December 2023. URL http://arxiv.org/abs/2309.07597. arXiv:2309.07597 [cs].

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. WizardLM: Empowering Large Language Models to Follow Complex Instructions, June 2023a. URL http://arxiv.org/abs/2304.12244. arXiv:2304.12244 [cs].

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the Tool Manipulation Capability of Open-source Large Language Models, May 2023b. URL https://arxiv.org/abs/2305.16504v1.

Rui Yang, Lin Song, Yanwei Li, Sijie Zhao, Yixiao Ge, Xiu Li, and Ying Shan. GPT4Tools: Teaching Large Language Model to Use Tools via Self-instruction, May 2023. URL http://arxiv.org/abs/2305.18752. arXiv:2305.18752 [cs].

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Ren Kan, Dongsheng Li, and Deqing Yang. Easytool: Enhancing llm-based agents with concise tool instruction. *arXiv preprint arXiv:2401.06201*, 2024.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. When language model meets private library, 2022. URL https://arxiv.org/abs/2210.17236.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large Language Models Meet NL2Code: A Survey. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7443–7464, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.411. URL https://aclanthology.org/2023.acl-long.411.

Kechi Zhang, Huangzhao Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. Toolcoder: Teach code generation models to use api search tools, 2023a. URL https://arxiv.org/abs/2305.04032.

Peitian Zhang, Shitao Xiao, Zheng Liu, Zhicheng Dou, and Jian-Yun Nie. Retrieve Anything To Augment Large Language Models, October 2023b. URL http://arxiv.org/abs/2310.07554. arXiv:2310.07554 [cs].

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A Appendix

A.1 Ethical and Considerations

Developers and other kind of end-users that plan to use API Pack for LLM fine-tuning should consider the following factors:

- **API Evolution and Validity:** APIs included in API Pack may evolve over time, potentially rendering some data outdated. Users should be aware that fine-tuning models on static API datasets might introduce deprecated or incorrect API calls. We recommend periodically updating the dataset or employing retrieval-augmented generation techniques to ensure the model has access to the most recent API specifications.
- **Hallucination:** API Pack's code (API calls) was not generated via an LLM. Thus, there is no risk of code errors in the dataset due to LLMs hallucination. That said, hallucination is a general risk associated with the use of LLMs. Therefore, code generated with LLMs that were fine-tuned with API Pack or any other code dataset must be always verified by a developer.
- **Data Ownership and License Information:** The OAS files used to build API Pack come from public and official API data hubs. API data owners published these files on those websites themselves. That said, if a legitimate API code owner requires their data to be removed from API Pack we will do so. Regarding API's license, only few OAS files contained this information. We are committed to tracking down complete and accurate license information for all APIs included in API Pack in a future version of the dataset.
- **Sensitive Information:** There is no risk of sensitive information being released as API Pack code data was sourced from API documentation that does not include real argument values. Instead, placeholders indicate the need for an argument value (i.e., REPLACE_BASIC_AUTH). In a real application, the users can provide these values through a conversational AI interface (i.e., code assistant).

A.2 Broader Impact

API Pack enables the integration of code LLMs specialized in API call generation into software development processes, which could have significant broader impacts:

- **Enhanced Software Productivity:** The presented advancements promise significant software development acceleration by automating routine coding tasks. This could enable faster iterations, reduced costs, and potentially higher-quality products. However, reliance on auto-generated code risks reduced developer accountability and control. The oversight of human eye to ensure accuracy is still required.
- **Sociotechnical Considerations and Responsible Innovation:** Although intelligent coding tools enhance productivity, sociotechnical concerns exist regarding job displacement, ownership, and digital inequality. We design API Pack to leverage the creation of tools that increase software productivity by keeping humans in the loop, as developers still decide whether or not using the API calls generated with models fine tuned on API Pack. Regarding code ownership, we consider API Pack not to have a negative effect, as data is sourced from documentation that is already publicly available. Towards reducing digital inequality, we have decided to open-sourced our dataset, code, and models.

A.3 Data Filtering at each Stage of the Pipeline

Table 7 shows the number of instances from each API source at different stages of our data filtering pipeline. Detailed description for each stage can be found in Section 3.

Table 7: Data Filtering Progress

Source/Instances	Before Data Validation	After Removing Invalid API calls	After Removing Instances without Good Instructions
IBM API Hub	27,635	17,712	17,206
APIs Gurus	500,160	499,250	495,533
Swaggerhub	351,756	351,756	345,765
RapidAPI	274,014	273,388	270,095
Total	**1,153,565**	**1,142,106**	**1,128,599**

A.4 Data Spliting for Level 1, Level 2, and Level 3 Testing

To split the data, we first copy the total dataset into total_training and calculate the occurrence of each API. For the first level of testing data, select up to 1,000 data points from the first 20,000 entries where instruction_test is not "None" and the API appears more than four times. For the second level, iterate through the first 20,000 entries, selecting data points with APIs not yet selected, until 1,000 unique APIs are collected, then remove these from the training dataset. For the third level, from the 80,000th entry onwards, select data points with APIs not used in Level 2, collecting up to 1,000 data points, and remove data points with these APIs from the training dataset.

A.5 Hyperparameters for Training

We fine-tune the models using the HuggingFace Transformers library on a cluster consisting of 1 node with 8 NVIDIA H100 80GB GPUs, with Fully Shared Data Parallelism (FSDP). Techniques such as mixed precision, gradient checkpointing, and AdaFactor optimizer are used to improve training efficiency. The key hyperparameters are summarized in Table 8.

A.6 Testing Pipeline

Figure 5 outlines our testing pipeline for 0-shot and few-shot learning. The retriever uses an embedding model with cosine similarity to find Top-k semantically similar examples from the example collection. A reranker then refines and re-orders these examples. The LLM generates the appropriate API call based on the user input and retrieved examples. For 0-shot inference, no examples are included in the prompt. For few-shot learning, we use a 3-shot approach, incorporating the three most relevant examples. The prompt template is in Listing 5.

Table 8: Hyperparameters for Training

Hyperparameter Name	Value
Learning rate	2×10^{-5}
Batch size	128
Max seq length	4096
Number of epochs	2
Warmup ratio	0.03

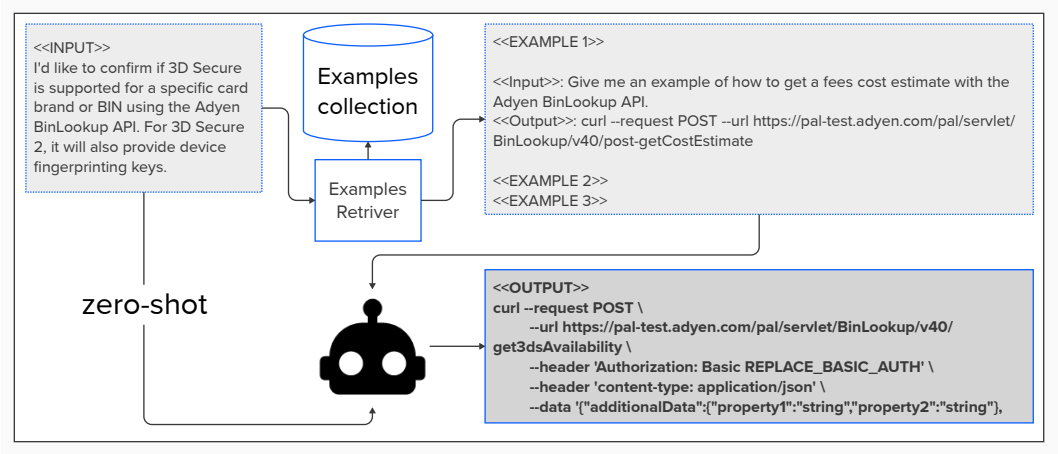

Figure 5: Testing pipeline.

A.7 API DB INSTANCE

Figure 6 shows and the structure of an API DB instance. We used openapi-snippet [7], an open-source package that takes as input an OpenAPI v2.0 or v3.0.x specification file (OAS file) and translates it into an HTTP Archive 1.2 request object, to generate API calls. We generated API calls (api_call) in 10 different programming languages (cURL, libcurl, java, node, python, go, ruby, php, swift, JavaScript) for RapidAPI, API Gurus, and the Swaggerhub. For the IBM Developer Hub API calls were extracted directly from the OAS files. We extracted API calls in eight different programming languages from this source (cURL, java, node, python, go, ruby, php, swift).

A.8 API PACK DATASET INSTANCE EXAMPLE

Figure 7 shows a fragment of an API Pack data instance. As data is sourced from public OAS files, personally identifiable and sensitive information in API calls is represented by anonymous strings (e.g., REPLACE_BASIC_AUTH), which indicates the need to pass this value as an argument in real execution. In this example, we only show one instruction for clarity, we represent the place for the other 4 instructions generated for this instance with the symbol ... .

[7] https://www.npmjs.com/package/openapi-snippet

```
{
    "api_name" : "The name of the API the endpoint belongs to",
    "api_description": "The description of the API the enpoint
     ↪  belongs to",
    "api_provider": "The API's provider name",
    "endpoint_name": "The name of the function to call",
    "functionality": "A brief description of endpoint's
     ↪  functionality",
    "description": "A long description of endpoint's functionality",
    "path": "The enpoint's name plus specific versioning as it
     ↪  appears in the API call's URL",
    "method": "HTTP method used in the API call (e.g., get, post, put,
     ↪  delete)",
    "api_call": "HTTP request code example to invoke an endpoint",
    "lang": "The programming language in which the API call is
     ↪  written",
}
```

Figure 6: Structure of an API DB instance

```
{
    "instruction_candidates": [
        {
            "idx": 1,
            "candidate": "I'd like to confirm if 3D Secure is supported for a specific
             ↪  card brand or BIN using the Adyen BinLookup API. For 3D Secure 2, it will
             ↪  also provide device fingerprinting keys.",
            "input_tokens_mean": -0.5497539341557909,
        },
        ...
    ],
    "api_call": "curl --request POST \
     --url https://paltest.adyen.com/pal/servlet/BinLookup/v40/get3dsAvailability \
     --header 'Authorization: Basic REPLACE_BASIC_AUTH' \
     --header 'content-type: application/json' \
     --data '{"additionalData":{"property1":"string","property2":"string"},
              "brands":["string"],
              "cardNumber":"string",
              "merchantAccount":"string",
              "recurringDetailReference":"string",
              "shopperReference":"string"}'"
    ...
}
```

Figure 7: Fragment of an API Pack instance in cURL.

A.9 Instruction Candidates

Figure 8 shows all the candidates generated for an instance with their respective id and input_tokens_mean.

A.10 Prompts

In this section, we share all the prompts used to create instructions. We also explain the process to craft each prompt. When in use, the values within the curly brackets would be replaced by the actual data.

A.10.1 Examples Refinement

We use Prompt 1 to refine the examples for instruction generation. We created three examples per API. Examples were manually refined by a human for the APIs from IBM API Hub and APIs Gurus. Then, we tasked an LLM with refining the examples for the other two sources.

Listing 1: Prompt for instruction example refinement

```
Your task is to refine and enhance a user query that involves a specific
    API. The original query you'll work with includes key details about
```

```
    "instruction_candidates": [
        {
            "idx": 1,
            "candidate": "I'd like to confirm if 3D Secure is supported
             ↪  for a specific card brand or BIN using the Adyen
             ↪  BinLookup API. For 3D Secure 2, it will also provide
             ↪  device fingerprinting keys.",
            "input_tokens_mean": -0.5497539341557909
        },
        {
            "idx": 2,
            "candidate": "To confirm if 3D Secure is enabled for a
             ↪  specific card brand or BIN number using the Adyen
             ↪  BinLookup API, you can make use of the
             ↪  `post-get3dsAvailability` endpoint. This API call will
             ↪  provide you with information about the availability of 3D
             ↪  Secure, along with device fingerprinting keys for 3D
             ↪  Secure 2 transactions. Let me know if you need assistance
             ↪  in forming the request or handling the response.",
            "input_tokens_mean": -0.5088001283229344
        },
        {
            "idx": 3,
            "candidate": "To confirm if a specific card brand or BIN
             ↪  supports 3D Secure and retrieves device fingerprinting
             ↪  keys for 3D Secure 2, please utilize the Adyen BinLookup
             ↪  API's post-get3dsAvailability endpoint.",
            "input_tokens_mean": -0.5554471563123543
        },
        {
            "idx": 4,
            "candidate": "To confirm if 3D Secure is supported for a
             ↪  specific card brand or BIN number using the Adyen
             ↪  BinLookup API, please make a POST request to the endpoint
             ↪  'post-get3dsAvailability'. This API will return whether
             ↪  3D Secure is enabled and for 3D Secure 2, it will also
             ↪  provide device fingerprinting keys.",
            "input_tokens_mean": -0.5466722401065375
        },
        {
            "idx": 5,
            "candidate": "To confirm if 3D Secure is supported by Adyen
             ↪  for a given card brand or BIN number, you can utilize the
             ↪  Adyen BinLookup API. Simply send a POST request to the
             ↪  post-get3dsAvailability endpoint with the required card
             ↪  details. If 3D Secure is available, the response will
             ↪  include device fingerprinting keys for 3D Secure 2.",
            "input_tokens_mean": -0.5726057469087047
        }
    ]
```


Figure 8: Examples of five instruction candidates generated for the same instance.

```
    the API's functionality, description, endpoint, and name. Focus on
    these essential aspects when revising the query:

1. **Integration of API Details:** Make sure the revised query includes
    relevant details about the API's functionality, description, and name
    , without directly mentioning the endpoint.

2. **Grammar and Syntax Correction:** Analyze the original query for
    grammatical mistakes such as improper verb forms (e.g., 'can'
```

```
    followed by 's' or 'es') or misplaced punctuation (like commas or
    colons). Correct these to improve clarity and professionalism.

3. **Relevance and Conciseness:** Eliminate any extraneous information
    from the original query. Strive for brevity while ensuring all
    critical details are included.

4. **User-Centric Rewrite:** Rework the query to reflect a user's
    perspective, focusing on their specific needs and how the API can
    address those needs.

For each query, you will receive:

- **Input:** An original user query with API details.
- **Your Task:** Revise the query based on the guidelines above.

Example for Practice:

### Input:
Functionality: Search Ecards
Description: Allows searching the inventory system for ecards.
Endpoint: searchECards
API: eCards Search API
User query to refine: "Please tell me how to searches ecards with the
    eCards Search API."

### Output (refined user query):
"Can you guide me on how to search for ecards using the eCards Search API
    ?"

Another Example:

### Input:
Functionality: KPI of realized sales
Description: Provides KPIs for documents issued, requires ACLs from /
    instances, and uses the 'typeKpi' parameter.
Endpoint: getKPIs
API: Blackbird Analytics
User query to refine: "I need to certificates shows kpis related to all
    documents issued through blackbird. to use it you must have pass a
    list of acls retrieved by /instances and specify which kpi using
      t y p e k p i   parameter."

### Output (refined user query):
"How can I access KPIs for documents issued by Blackbird Analytics, and
    what are the required 'typeKpi' parameters?"

Remember, the goal is to modify the user query to be clear, effective,
    and grammatically correct, fully showcasing how the user can leverage
     the specific API.

Now, here is your actual task:
### Input:
Functionality: {functionality}
Description: {description}
Endpoint: {endpoint}
API: {api_name}
User query to refine: {template generated instruction}

### Output (refined user query):
```

A.10.2 INSTRUCTION GENERATION

We use Prompt 2 to generate five instruction candidates for each API Pack instance. We crafted a custom prompt that considers the information a developer may provide to an LLM, while looking for an API call to complete a task.

Listing 2: Prompt for instruction generation

```
Your task is to create a user query that effectively utilizes a specific
    API. The API's functionality, description, and name will be provided
    to you. Your query should be designed in a way that makes the best
    use of this API's unique capabilities. When crafting your query,
    focus on:

1. **API Name Integration:** Clearly include the API's name in your query
     to ensure relevance.
2. **Specificity:** Replace broad or vague terms with precise, concrete
    details relevant to the API's purpose.
3. **Conciseness:** Keep your query as brief as possible while still
    fully conveying the needed information. Avoid unnecessary verbosity.
4. **Excluding API Endpoint:** Do not include the API's endpoint in your
    query; focus only on the user's need and how the API fulfills it.

Create a query that a user might realistically use when interacting with
    the given API. Think about typical scenarios or problems that the API
     is designed to solve and formulate your query accordingly.

Examples for practice:

###Input:
Functionality: {functionality}
Description: {description}
Endpoint: {endpoint}
API: {api_name}
###Output:
{output}

###Input:
Functionality: {functionality}
Description: {description}
Endpoint: {endpoint}
API: {api_name}
###Output:
{output}

###Input:
Functionality: {functionality}
Description: {description}
Endpoint: {endpoint}
API: {api_name}
###Output:
{output}

Remember, the goal is to demonstrate how a user would benefit from this
    specific API in a realistic scenario, using precise and clear
    language. Here is the actual task for you:

###Input:
Functionality: {functionality}
Description: {description}
Endpoint: {endpoint}
API: {api_name}
###Output:
```

A.10.3 BACK TRANSLATION

Prompt 3 seeks to re-generate the API information associated to an instruction by passing an instruction candidate as input. To create this prompt, we took Prompt 2 as reference and simply reversed the task. As a result of using this prompt for inference, we save the input-token-mean of the output return (the API information that was re-generated).

Listing 3: Prompt for instruction backtranslation

```
Your task involves a reverse-engineering process where you will analyze a
     user query to infer specific details about an API endpoint. Based on
     the given user query, you are expected to:

1. **Identify the Endpoint's Identifier:** Derive the endpoint identifier
     that aligns with the functionality implied by the user query.
2. **Determine Endpoint Functionality:** Interpret the user query to
    understand and describe the functionality of the endpoint.
3. **Describe the Endpoint:** Provide a detailed description of the
    endpoint based on the needs and context presented in the user query.
4. **Specify the API Name:** Identify and state the name of the API to
    which this endpoint belongs, as suggested by the user query.

Your response should clearly articulate these four elements (identifier,
    functionality, description, API name) in a manner that reflects an
    accurate understanding of the user query. Consider the query as a
    real-world scenario or problem that the endpoint is designed to
    address.

Examples for practice:

###Input:
{generated instruction}
###Output:
Functionality: {functionality}
Description: {description}
Endpoint: {endpoint}
API: {api_name}

###Input:
{generated instruction}
###Output:
Functionality: {functionality}
Description: {description}
Endpoint: {endpoint}
API: {api_name}

###Input:
{generated instruction}
###Output:
Functionality: {functionality}
Description: {description}
Endpoint: {endpoint}
API: {api_name}

The goal is to showcase your ability to connect a user's needs with the
    appropriate API endpoint, demonstrating an understanding of how the
    endpoint's features align with user requirements. Your response
    should be precise, insightful, and reflective of the query's
    implications. Here is the actual task for you:

###Input:
{generated instruction}
###Output:
Functionality: {functionality}
```

```
Description: {description}
Endpoint: {endpoint}
API: {api_name}
```

A.10.4 INSTRUCTION ANNOTATION

We use Prompt 4 to task an LLM with annotating the instruction candidates as good or bad. First, a human annotator manually reviewed 605 instructions to distill common error patterns among them. Then, we craft a prompt to detect these error patterns via an LLM. Note the prompt includes examples for 'good' and 'bad' instructions. These examples were extracted from the manual revision performed by the human annotator.

Listing 4: Prompt for instruction annotation

```
**Your Task**: Evaluate the provided instruction from an AI assistant and
     classify its quality as **Good** or **Bad** based on specific
    criteria.

**Criteria for 'Bad' Instruction**:
1. Contains multiple instructions instead of a single, clear directive.
2. Includes unnecessary additional text before or after the main
    instruction.
3. Fails to accurately use the specified API name and endpoint.

**Input Structure**:
You will receive an **INPUT** consisting of three elements:
1. An **instruction** generated by the AI assistant.
2. The **API name** related to the instruction.
3. The **API endpoint** relevant to the instruction.

**Output**:
Classify the instruction as **Good** or **Bad** with a concise
    justification.

**Examples**:

1. **Input**:
   - **Instruction**: "Create a new message in IBM Event Streams using
    the REST Producer API." "How do I format and send a message body via
    the IBM Event Streams REST Producer API?" "Can you help me construct
    a message using the IBM Event Streams REST Producer?" "Use the IBM
    Event Streams REST Producer API to send a message with specific data
    ." "What's the proper syntax for creating and sending a message
    through the IBM Event Streams REST Producer?"
   - **API Name**: IBM Event Streams REST Producer
   - **API Endpoint**: produceMessage
   - **Output**: Bad. This instruction is classified as **Bad** because
    it combines four separate instructions into one, each asking for
    different guidance related to the IBM Event Streams REST Producer API
    .

2. **Input**:
   - **Instruction**: "I'd like to send a new message to IBM Event
    Streams. How do I format the request body to ensure it is correctly
    processed using the IBM Event Streams REST Producer API?"
   - **API Name**: IBM Event Streams REST Producer
   - **API Endpoint**: produceMessage
   - **Output**: Good. This instruction is classified as **Good** because
     it provides a single, clear directive without additional text and
    correctly uses the API name.

3. **Input**:
   - **Instruction**: "Here's a possible user query utilizing the given
    API: 'Help me list all the bare metal servers in my Virtual Private
    Cloud account using the Virtual Private Cloud API.'"
```

```
   - **API Name**: Virtual Private Cloud API
   - **API Endpoint**: list_bare_metal_servers
   - **Output**: Bad. This instruction is classified as **Bad** because
    it contains unnecessary introductory text ("Here's a possible user
    query utilizing the given API:") before the actual instruction.

4. **Input**:
   - **Instruction**: "How do I retrieve a list of all bare metal servers
     in my region using the Virtual Private Cloud API?"
   - **API Name**: Virtual Private Cloud API
   - **API Endpoint**: list_bare_metal_servers
   - **Output**: Good. This instruction is classified as **Good** because
     it is a singular, straightforward instruction without extra text and
     appropriately uses the API name.

**Your Current Task**:

**Input**:
- **Instruction**: {candidate}
- **API Name**: {api_name}
- **API Endpoint**: {endpoint}

**Output**:
```

A.10.5 Few-shot Fine-tuning and Evaluation

This is the prompt template used for 3-shot-tuning to post-process API Pack data, as well as for 3-shot random, 3-shot retrieved, and 3-shot retrieved & re-ranked inference.

Listing 5: Prompt for few-shot fine-tuning and evaluation

```
**api_description**:{api_description}
**lang**:{programming language}

Given the following examples:

**instruction**
{instruction}
**output**
{api_call}

**instruction**
{instruction}
**output**
{api_call}

**instruction**
{instruction}
**output**
{api_call}

Your actual task:
**instruction**
{instruction_test}
**output**

### ASSISTANT:
```