Benchmarking Web API Integration Code Generation

Daniel Maninger
Technische Universität Darmstadt
Darmstadt, Germany
Hessian Center for Artificial Intelligence (hessian.AI)
Darmstadt, Germany
daniel.maninger@tu-darmstadt.de

Leon Chemnitz
Pariton AI
Berlin, Germany
leon.chemnitz@pariton.ai

Amir Molzam Sharifloo
Technische Universität Darmstadt
Darmstadt, Germany
amir.molzam@tu-darmstadt.de

Jannis Brugger
Technische Universität Darmstadt
Darmstadt, Germany
Hessian Center for Artificial Intelligence (hessian.AI)
Darmstadt, Germany
jannis.brugger@tu-darmstadt.de

Mira Mezini
Technische Universität Darmstadt
Darmstadt, Germany
Hessian Center for Artificial Intelligence (hessian.AI)
Darmstadt, Germany
National Research Center for Applied Cybersecurity ATHENE
mezini@cs.tu-darmstadt.de

Abstract

API integration is a cornerstone of our digital infrastructure, enabling software systems to connect and interact. However, as shown by many studies, writing or generating correct code to invoke APIs, particularly web APIs, is challenging. Although large language models (LLMs) have become popular in software development, their effectiveness in automating the generation of web API integration code remains unexplored. In order to address this, we present a dataset and evaluation pipeline designed to assess the ability of LLMs to generate web API invocation code. Our experiments with several open-source LLMs reveal that generating API invocations poses a significant challenge, resulting in hallucinated endpoints, incorrect argument usage, and other errors. None of the evaluated open-source models were able to solve more than 40% of the tasks.

CCS Concepts

• **Software and its engineering → Automatic programming**; • **Computing methodologies → Neural networks**; • **Information systems → RESTful web services**.

Keywords

artificial intelligence, software engineering, large language models, code generation, web APIs

ACM Reference Format:
Daniel Maninger, Leon Chemnitz, Amir Molzam Sharifloo, Jannis Brugger, and Mira Mezini. 2025. Benchmarking Web API Integration Code Generation. In *Proceedings of 2nd ACM International Conference on AI-powered Software, Benchmark & Dataset Track (AIware '25).* ACM, New York, NY, USA, 19 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Web APIs provide services and functionality through a standardized, HTTP-based interface. Developers leverage them to rapidly create applications (powered by so-called *API integration code*) that seamlessly integrate with a wide range of online services. The market surrounding API integrations is experiencing explosive growth [27, 31].

However, writing correct API integration code is a tedious and challenging task. Developers need to understand and correctly utilize a significant amount of information to be able to write an *API invocation* in compliance with its specification [33, 35]. Moreover, the task of ensuring correct API usage is further complicated by the fact that APIs evolve and change over time.

Large language models (LLMs) have great potential to boost productivity in software development thanks to their ability to assist developers, e.g., by automatically generating program code from natural language descriptions [4]. However, LLMs may also hallucinate and make mistakes, raising concerns about correctness [8, 44], security [29, 30], and general quality [13] of LLM-generated code.

Research gap. While the capabilities of LLMs have been investigated for many different software engineering tasks, including library (i.e., non-web) API invocations [51], their aptitude for web API integrations appears to be rather unexplored (cf. also related work in Section 4). The question is: Given the pivotal role of web APIs, can we unlock the potential of LLMs for streamlining and accelerating the development processes of software that involves web API integration? As a first step towards answering this question, we need to understand how well existing LLMs support that task. To this end, we formulate the following research question:

> **RQ:** *How correct is LLM-generated web API invocation code, and what types of errors are commonly present in such code?*

Listing 1: General structure of an API invocation using Axios and JavaScript.

```
const axios = require('axios');
axios.<method>('https://server.com/path/to/endpoint/arg1', {
    arg2: 'request body parameters'
}, {
    headers: { arg3: 'header parameters' },
    params: { arg4: 'query parameters' }
});
```

Methodology. We investigate the stated research question through the following task: Given a short program context and a code comment that describes a desired operation on a given web API in natural language, the LLM has to complete the code by implementing the operation in compliance with the API's specification.

To arrive at an answer, we perform the following steps. First, we create a dataset of API invocation tasks and expected outcomes. Using a powerful state-of-the-art LLM, we generate the initial version of the dataset and subject it to a thorough manual review and correction process. Second, we implement an evaluation pipeline that allows us to safely execute web API invocation code and evaluate the resulting requests to obtain fine-grained correctness metrics. For this purpose, we develop a novel evaluation methodology. Third, we use our dataset and evaluation pipeline to evaluate a comprehensive set of LLMs, including models from the StarCoder [23, 26], DeepSeek-Coder [5, 15], Qwen-Coder [19], (Code) Llama [9, 36], and GPT families.

Scope. Since web APIs can be used in many ways, we need to be more specific and focus our research specifically on the following setup: We target real-world web APIs that adhere to *OpenAPI*[1], the most widely used specification standard for web APIs [39]. We generate code in *JavaScript*, one of the most popular programming languages [41], and use *Axios*[2], the leading JavaScript library for invoking web APIs [14], which allows explicit configuration of all components of an HTTP requests.

The general structure of the API invocations we consider is shown in Listing 1. An invocation comprises an HTTP method (GET, POST, etc.), a URL with server address and (potentially parameterized) path to the endpoint, a request body, a request header, and query parameters. Such web API invocations have some unique challenges compared to function calls in a local codebase: (1) Operations are identified by the combination of a method and a long URL string, not just a simple function name; (2) there is not just one but multiple argument lists and the number of available arguments can be much higher while many arguments have complex, nested data types; and (3) web API endpoints are documented externally, limiting accessibility for LLMs, and the used specification format is much more complex than, e.g., that of documentation comments used for local functions.

Evaluation results. Our study reveals that, without any in-context information about the APIs, most LLMs demonstrate some understanding of the assigned tasks, show familiarity with the general structure of invocations in JavaScript/Axios, and appear to have memorized portions of the APIs' specifications. However, even the best open-source models generate correct invocations only 40% of the time. In particular, they frequently hallucinate endpoint URLs (up to 39%) and parameter names (up to 31%).

Contributions. In summary, our contributions are as follows:

A first-of-its-kind dataset of web API invocation tasks across four real-world APIs with nearly 400 pairs of tasks and expected outcomes, which can be used for execution-based correctness analysis. → Section 2.1

An open-source evaluation pipeline for automatically and safely evaluating the performance of code generation models on web API invocation tasks using fine-grained metrics. → Section 2.2 – Section 2.4

Novel empirical insights on the (in)ability of LLMs to generate correct API invocation code. → Section 3

All source code and data referred to in this paper is available in our artifact[3]. Upon acceptance of the paper, the entire project will be published under an open-source license.

2 Benchmark Design

Our evaluation pipeline, depicted in Figure 1, consists of four stages: (1) *dataset creation*, (2) *code generation*, (3) *code execution*, and (4) *correctness analysis*. They are detailed in the following subsections.

2.1 Dataset Creation

We conceive of our dataset as a collection of triples (a, t, c), where a is the name of the targeted API, t is a task, given as an unambiguous natural language description of a request to a, and c is a so-called request *configuration*, which captures all properties an API request must have in order to solve t—it is the key to measuring functional correctness. Specifically, c contains the expected URL, HTTP method, and arguments in the request body, header, and query locations. In this paper, we use the term *endpoint* to refer to the combination of URL and (HTTP) method.

To our knowledge, no dataset with the described structure exists. To create a new one, two general approaches exist: sourcing data from the real world or synthesizing it. We first explored the former option by searching public GitHub repositories for suitable JavaScript/Axios code[4]. We dropped this option and resorted to the latter for two reasons. First, we found that real-world code rarely includes sufficient information (e.g., comments or in-code documentation) to serve as an unambiguous task description t. Second, the resulting dataset would have been incomplete, not covering all endpoints of a given API.

Instead, we opted for a synthetic approach, using an LLM Gen and API specifications s to generate task–configuration pairs for our dataset: $\text{Gen}(a, s) = (t, c)$. To ensure our dataset represents real-world scenarios, we chose APIs of popular services across different domains—Asana, Google Calendar, Google Sheets, and Slack—which are specified following OpenAPI, the most commonly used industry standard [39]. As Gen, we used Gemini 1.5 Pro[5] for its very large context window (2M tokens), given that OpenAPI

[1]https://www.openapis.org/
[2]https://axios-http.com/

[3]https://zenodo.org/doi/10.5281/zenodo.13758414 (Note that the artifact also includes work-in-progress implementations for future work, which can be ignored.)
[4]Using BigQuery (https://cloud.google.com/bigquery/) for this purpose.
[5]https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/

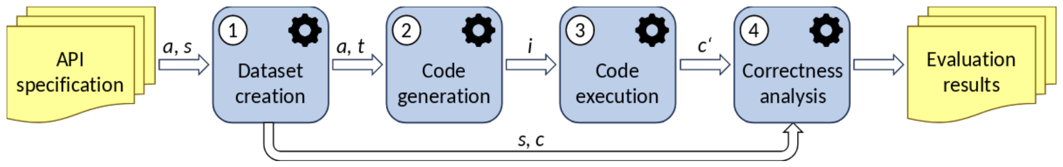


Figure 1: Experimental setup to evaluate LLMs' capabilities in generating API invocation code. (1) Based on an API a and its specifications s, an API invocation tasks t and corresponding correct request configurations c are created. (2) For each t, the LLM under evaluation generates an API invocation i. (3) i is executed in a controlled environment, yielding a request configuration c'. (4) c' is compared to c and validated against s to obtain various metrics, shown in Table 2.

Table 1: Number of endpoints per HTTP method for each API. Each endpoint corresponds to one sample in our dataset.

API	Total	POST	GET	PUT	DELETE	PATCH
Asana	167	61	79	14	13	0
Google Calendar	37	14	11	4	4	4
Google Sheets	17	12	4	1	0	0
Slack Web	174	95	79	0	0	0
Total	395	182	173	19	17	4

specifications can be very long (e.g., Slack's contains over 20,000 lines of text). Together with the full specification, we provided the model with very detailed (but mostly non-API-specific) instructions on how to construct t and c in a zero-shot prompt to avoid pitfalls of few-shot prompting [34] and increase generalizability. The prompt is provided in the appendix. In total, we synthesized 395 samples—one for each endpoint of the selected APIs. A detailed breakdown is shown in Table 1.

To ensure the validity of the AI-generated data, we curated all samples with automated and manual checks. The automated checks caught any inconsistencies between the generated configurations c and the specifications s. The 9 samples that failed these checks were manually inspected and corrected. Afterward, we manually checked all 395 samples to catch remaining issues that cannot be checked automatically, especially underspecification and ambiguities in the tasks t. Concretely, we checked (1) that t is well-formulated and solvable; (2) that c actually solves t; and (3) that every argument in t is reflected in c and vice versa. 58 samples had one or multiple issues and were fixed by adjusting t and/or c.

Listing 2 shows a simplified example from our dataset, with the API a being *Google Calendar*. The task t is given as a comment, followed by the implementation the model has to generate. Listing 3 shows the corresponding configuration c, which contains key–value pairs specifying the correct HTTP method, URL, and parameters—in this case, there are only parameters in the request body (“`data`”).

2.2 Code Generation

We take each pair of API a and task t from our dataset and let the model under evaluation M generate an API invocation i: $\mathrm{M}(a, t) = i$. The prompt instructs M to solve t by completing the starter code that is given below it. We specify a in order to disambiguate t as otherwise a common task like “create a new user ...” could refer to many different APIs. The prompt is provided in the appendix.

Listing 2: Simplified task from our dataset with correct implementation.

```
// Create a new secondary calendar named "Example Calendar" with
      time zone "America/Los_Angeles".
const axios = require('axios');
axios.post('https://www.googleapis.com/calendar/v3/calendars', {
    summary: 'Example Calendar',
    timeZone: 'America/Los_Angeles'
});
```

Listing 3: Request configuration corresponding to the task in Listing 2.

```
{
  "method": "post",
  "url": "https://www.googleapis.com/calendar/v3/calendars",
  "data": {
    "summary": "Example Calendar",
    "timeZone": "America/Los_Angeles"
  }
}
```

We also include some clarifications in the prompt, meant to avoid instances of misalignment and recurring error patterns we observed in preliminary experiments. The starter code always includes t as a line comment (e.g., `// Create a new secondary calendar ...`), besides the required imports.

We conduct our experiments with two different starter code variants—we call them *setups*—that serve complementary purposes. *Full completion* includes the beginning of API invocation (`axios.`). By fixing the code up to this position, we ensure that an Axios call is actually generated and can be evaluated. In this setup, we can assess the ability of the model to identify the correct endpoint for t and to solve the task as a whole. *Argument completion* additionally includes the correct method and URL (e.g., `axios.post('https://www.googleapis.com/calendar/v3/calendars',`). In this setup, we can evaluate whether the model is able to use a given endpoint in compliance with a. Also, this setup resembles real-world scenarios inside an IDE where the user has already implemented a part of the solution and wants it to be completed by the AI.

Note that the code generation task described here is considerably different from the dataset generation task from Section 2.1. In the latter, Gemini had access to the full API specification and very detailed instructions, making this a task about processing in-context information. The models under evaluation, on the other hand, have to rely solely on memorized knowledge about the APIs, as we want to find out how well they can recall these details from their training

Table 2: Evaluation metrics for correctness and specification-compliance of API invocations.

Metric	Description
Correct implementations	Generated executable code matches the ground-truth configuration
Illegal implementations	Generated code contains at least one violation of the API specification
Correct URLs	Generated URL matches the ground-truth URL
Illegal URLs	Generated URL is not defined in the API specification
Correct methods	Generated HTTP method matches the ground-truth HTTP method
Illegal methods	Generated HTTP method is not defined for the generated URL in the API specification
Mean argument precision	Probability that the generated arguments are correct
Mean argument recall	Probability that the correct arguments are generated
Illegal arguments	Generated arguments are not permitted for the generated API endpoint
Mean argument value conditional accuracy	Probability that an argument value is correct if the argument name is correct

data. This is a relevant concern as context length is limited, and it is often not desirable to fill the context with too much information.

2.3 Code Execution

Our approach to evaluating LLMs' capabilities in generating API invocation code is based on *functional correctness* rather than syntactic similarity to a reference implementation, as the latter is prone to misjudgments [12]. To measure functional correctness, we need to execute the generated code and assess its outcome, analogous to unit testing [3].

When executing i, we need to address challenges unique to our setting. First, direct execution of generated code is unsafe as it could send arbitrary requests to external servers. Second, we need to capture and evaluate the configuration c' of the API request implemented by i, which is a *side effect* and not the return value of i. Another common challenge when evaluating code generation is *excess code* [16], i.e., code produced when the model terminates too late.

To address these challenges, we implement a controlled execution environment Mock, which intercepts requests before transmission and then serializes and returns their configurations: $\text{Mock}(i) = c'$. We address the *excess code* issue in Mock by heuristically truncating the generated code after the API invocation, preventing the execution of unnecessary, incomplete, broken, or harmful code.

2.4 Correctness Analysis

We compare the configuration c', captured in the previous step, with the ground-truth c from our dataset: $c' \overset{?}{=} c$. However, as web API requests contain multiple components—URL, method, and parameters in different locations (cf. Listing 1)—only looking for exact matches is insufficient. Instead, we compare the configurations in an element-wise manner as well as in aggregated forms to facilitate a fine-grained evaluation. Additionally, we validate c' against the API specification s to distinguish between endpoints and arguments that are undesired but specification-compliant and ones that are actually illegal. The key metrics we calculate are explained in Table 2 and additional ones are provided in the appendix. Note that some metrics apply only to one of the setups (full/argument completion), as mentioned in Section 2.2.

3 Evaluation

Using our dataset and evaluation pipeline, we study the performance of 21 state-of-the-art open-source LLMs on web API invocation code generation. We primarily evaluated models that have undergone fine-tuning on code data—CodeT5+ [46] (16B), StarCoder [23] (15.5B), StarCoder2 [26] (15B), DeepSeek-Coder [15] (6.7B, 7B, 33B), DeepSeek-Coder-V2 [5] (16B), Qwen2.5-Coder [19] (0.5B, 1.5B, 3B, 7B, 14B, 32B), and CodeLlama [36] (7B, 13B, 70B)—and additionally Llama 3.1 [9] (8B, 70B). These models were selected due to their frequent appearance in coding leaderboards[6]. To obtain an upper bound on performance, we also evaluated OpenAI's commercial GPT-4o[7] and GPT-4o mini[8] models. For a lack of space and for more visual clarity, we show only a subset of these models in our plots and tables, but our following analysis considers all of them. Comprehensive result tables are provided in the appendix.

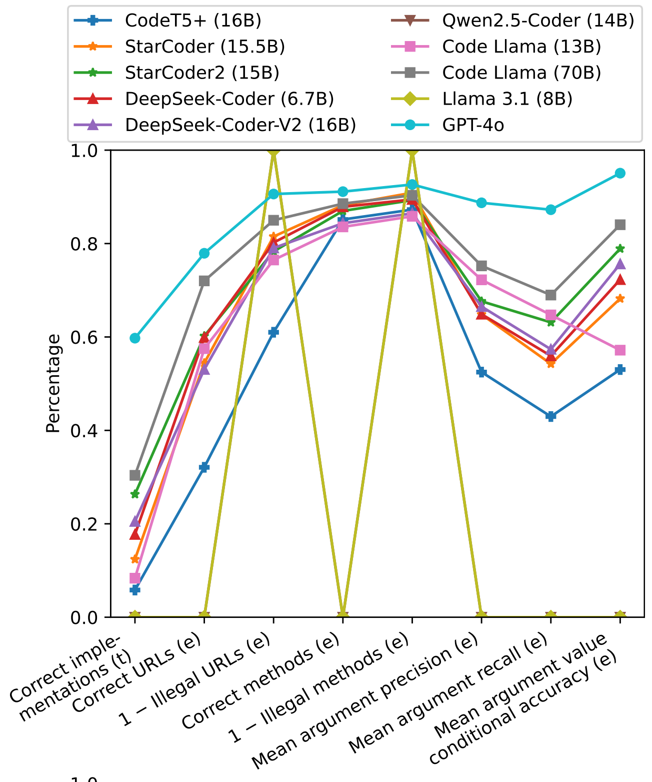
Throughout this paper, we use the notation *(t)* to indicate that a metric is calculated based on the *total* amount of samples in the dataset and *(e)* to indicate that it is calculated based on only the samples that resulted in *executable* code[9]. We do this because our evaluation method can only assess the correctness of individual request components if the generated code can be executed, and would otherwise classify them as incorrect by default. The distinction between (t) and (e) allows us to differentiate between a model's ability to generate *complete* and *executable* code (t), and a model's ability to generate code that *complies with the specification* and *solves the given task*, provided that it is executable (e)[10].

3.1 Results

Figure 2 shows key performance metrics (explained in Table 2) for one representative[11] of each model family plus the best-performing open-source model for full and argument completion, respectively. The corresponding numbers are provided in Tables 3 and 4. We used greedy decoding for all experiments.

Full completion. Across all evaluated models, the achieved correctness (*correct implementations (t)*) ranges from 0% for the Qwen2.5-Coder and Llama 3.1 models up to 60% for GPT-4o, which also dominates on all other metrics. The best open-source model is Code Llama (70B) at 30% followed by StarCoder2 at 26%. In all cases

[6]E.g., https://evalplus.github.io/leaderboard.html
[7]https://openai.com/index/hello-gpt-4o/
[8]https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/
[9]More specifically, the calculation for (t) and (e) values is identical except for a filtering step to remove non-executable samples before calculating the latter.
[10]For illustration why this distinction is necessary, consider a model M that predicts the correct endpoint 100% of the time but introduces syntax errors in the request arguments 50% of the time. When looking at (t) metrics only, we would wrongly conclude that M predicts endpoints correctly only 50% of the time.
[11]With a size as close as possible to 15B for a fair comparison.



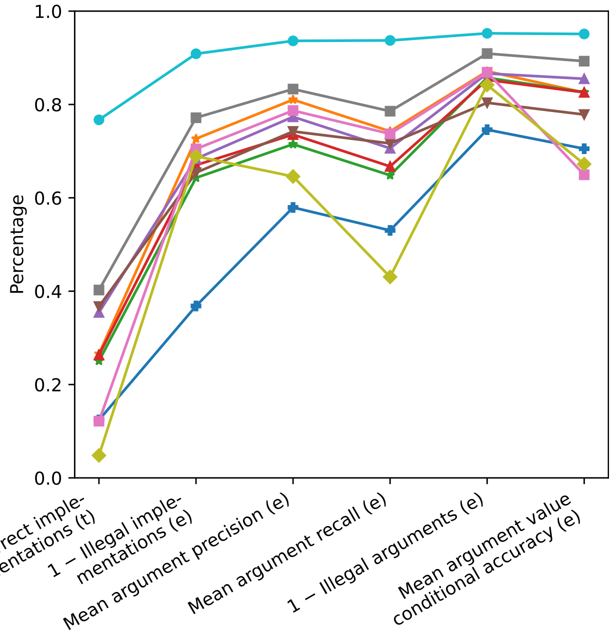


Figure 2: Performance comparison between models for *full completion* (top) and *argument completion* (bottom).

with 0% correctness, (almost) none of the generated implementations are executable—since also all other metrics for these models become zero, we omit reporting their numbers in the following. For the remaining models, the executability is 94% and higher.

Most open-source models correctly generate over 80% of HTTP methods (e). The rate of correct URLs (e) varies strongly between 32% and 72%, but 14% to 39% of URLs are illegal (e). All models have higher precision (e) than recall (e) for arguments, ranging from 51% to 75% and from 33% to 69%, respectively. If an argument name was predicted correctly, the argument value was correct 53%–84% of the time (e). For all of these metrics, Code Llama (70B) is the strongest open-source model, even surpassing GPT-4o mini. DeepSeek-Coder (7B) and CodeT5+ (16B) perform the worst across individual metrics.

Argument completion. Compared to full completion, almost all performance metrics are better for argument completion. The rate of correct implementations (t) ranges from 8% for DeepSeek-Coder (7B) to 40% for Code Llama (70B), which is again the strongest open-source model. The best performing model overall is again GPT-4o at 77% correctness. The models that struggled to generate any executable code before now achieve similar executability rates as the other models. Qwen2.5-Coder (14B) and (32B) are now even among the better-performing models.

Overall, for all metrics except correct implementations (t), we observe a smaller variance between the open-source models and an even clearer performance gap between them and GPT-4o. Since in the argument completion setup, the correct endpoint is provided to the models, we can now evaluate their ability to generate argument lists without violating the API specification. We observe that 6% to 31% of generated arguments are illegal (e).

3.2 Discussion

In summary, all tested open-source LLMs struggle with generating correct API invocations. While they often generate partially correct implementations—indicating that the models have had training exposure to API specifications or examples from which they memorized usage patterns of the respective APIs—the models fail to combine individual patterns to fully correct API invocations. The full completion setup shows that the models often do not find the correct endpoint to use. But even when the correct endpoint is provided in the argument completion setup, the models still often fail to stay consistent with the given endpoint and its defined parameters.

The models perform comparatively well in predicting the HTTP method, which is expected since there is only a small set of available methods, with GET and POST being by far the most common ones. In contrast, URLs and arguments are challenging to predict due to their unrestricted nature. The high argument precision can be attributed to the `Authorization` header argument, which is required my most endpoints and therefore easy to predict. The relatively high argument value conditional accuracy indicates that identifying the correct argument is harder for the models than assigning the correct value to it once it is found.

An analysis of the generated implementations shows that those models that got nearly zero implementations correct in the full completion setup failed to understand the task given to them and refused to properly continue the starter code, leading to incomplete and thus non-executable code. It may be possible to solve this issue through prompt engineering, but we preferred to use a single prompt consistently for all models.

Table 3: Evaluation results for *full completion.*

	CodeT5+ (16B)	StarCoder (15.5B)	StarCoder2 (15B)	DeepSeek-Coder (6.7B)	DeepSeek-Coder-V2 (16B)	Qwen2.5-Coder (14B)	Code Llama (13B)	Code Llama (70B)	Llama 3.1 (8B)	GPT-4o
Correct implementations (t)	0.06	0.12	0.26	0.18	0.21	0.00	0.08	0.30	0.00	**0.60**
Correct URLs (e)	0.32	0.54	0.60	0.60	0.53	0.00	0.58	0.72	0.00	**0.78**
Illegal URLs (e)	0.39	0.18	0.22	0.20	0.21	**0.00**	0.24	0.15	**0.00**	0.09
Correct methods (e)	0.85	0.88	0.87	0.88	0.84	0.00	0.84	0.89	0.00	**0.91**
Illegal methods (e)	0.13	0.09	0.11	0.11	0.14	**0.00**	0.14	0.10	**0.00**	0.07
Mean argument precision (e)	0.52	0.65	0.68	0.65	0.67	0.00	0.72	0.75	0.00	**0.89**
Mean argument recall (e)	0.43	0.54	0.63	0.56	0.57	0.00	0.65	0.69	0.00	**0.87**
Mean arg. val. cond. acc. (e)	0.53	0.68	0.79	0.72	0.76	0.00	0.57	0.84	0.00	**0.95**

Table 4: Evaluation results for *argument completion.*

	CodeT5+ (16B)	StarCoder (15.5B)	StarCoder2 (15B)	DeepSeek-Coder (6.7B)	DeepSeek-Coder-V2 (16B)	Qwen2.5-Coder (14B)	Code Llama (13B)	Code Llama (70B)	Llama 3.1 (8B)	GPT-4o
Correct implementations (t)	0.12	0.27	0.25	0.26	0.35	0.37	0.12	0.40	0.05	**0.77**
Illegal implementations (e)	0.63	0.27	0.36	0.33	0.32	0.35	0.30	0.23	0.31	**0.09**
Mean argument precision (e)	0.58	0.81	0.71	0.74	0.77	0.74	0.79	0.83	0.65	**0.94**
Mean argument recall (e)	0.53	0.74	0.65	0.67	0.71	0.72	0.74	0.79	0.43	**0.94**
Illegal arguments (e)	0.25	0.13	0.14	0.15	0.13	0.20	0.13	0.09	0.16	**0.05**
Mean arg. val. cond. acc. (e)	0.71	0.83	0.83	0.83	0.86	0.78	0.65	0.89	0.67	**0.95**

An additional finding is that larger models are not always better. In multiple model families (DeepSeek-Coder, Qwen2.5-Coder, Code Llama), we observe that the medium-sized variant performs worse than both the smaller and larger variants. Lastly, breaking down the evaluation results by API (cf. tables in the appendix), we see that each model is particularly good or bad at different APIs, which likely results from the prevalence of the respective APIs in the model's training data.

> **RQ:** *How correct is LLM-generated API invocation code, and what types of errors are commonly present in such code?*
> **Answer:** While LLMs are able to generate partially correct API invocations, the overall result is incorrect most of the time. The types of errors commonly present are manifold and range from selecting the wrong endpoint to leaving out required arguments or hallucinating entirely illegal arguments.

3.3 Limitations and Threads to Validity

1) Real-world API integration code is diverse and complex. To be able to evaluate API invocations automatically, we had to apply some restrictions and simplifications when creating the tasks in our dataset, as described in Section 2.1. For instance, there is limited program context before the API invocation. Additionally, we evaluate only the correctness of the outgoing request, not how the incoming response is handled. These limitations do, however, not interfere with our investigation of LLM's ability to recall information about correct API usage—if this information is memorized, they should be able to recall it in our evaluation *and* real-world settings.

2) The quality of the dataset, evaluation, and constraints depend on the quality of the API specifications. We encountered and manually fixed several errors in the real-world specifications. Moreover, the specifications sometimes contain usage constraints that are only explained in a parameter's free-form textual description and thus not captured by our automated correctness analysis.

3) Despite thorough vetting, our dataset might contain faulty samples that could introduce noise into the evaluation results. We also noticed that the synthetic tasks produced by Gemini 1.5 Pro tend to use optional parameters rather sparingly and often use placeholder values instead of realistic example values, limiting the transferability of findings from our benchmark to real-world API invocation tasks. Moreover, the results for some models may not generalize to other APIs, given the limited number of APIs included in our dataset. As models can be very sensitive to prompt variations, the performance of each model can almost certainly be optimized by customized prompt engineering.

4 Related Work

Differentiation of API Invocation Settings. Since APIs are such a universal concept, it is important to distinguish between different settings in which APIs are being used and understand their different characteristics when discussing related work. We divide APIs roughly into the following categories: (1) *general web APIs* (this work), (2) *domain-specific web APIs*, (3) *SDK-wrapped web APIs*, (4) *local function APIs*, and (5) *tool APIs for AI agents*.

1) Our work targets general web API invocations, which involve explicitly configuring all components of an HTTP request, i.e., an endpoint URL, HTTP method, and multiple request argument lists

with oftentimes complex nested data types. To our knowledge, we are the first to explore and to propose a method for evaluating the capabilities of LLMs in generating code with such specific and challenging characteristics.

2) Su et al. [42, 43] and Hosseini et al. [18] investigated natural language interfaces to domain-specific web APIs following the Open Data Protocol[12] (OData). Their work has a narrow focus on data-centric applications (mostly within the Microsoft ecosystem) and an SQL-like syntax. In contrast, our benchmark targets OpenAPI, a very general and flexible API standard, making the task more open-ended and challenging. Also, instead of ad-hoc API queries by a user, we strive for production-ready API integration code implemented in JavaScript.

3) Works like Dialog2API [38] and CloudAPIBench [20] focused on SDK-wrapped web API invocations (e.g., for AWS), treating web APIs like local libraries. Both approaches do not generalize due to SDK heterogeneity. In contrast, our benchmark relies on a unified RESTful interface for *any* web API. This eliminates SDK dependency, enabling direct assessment of models' core API comprehension rather than SDK-specific knowledge. By standardizing interactions across diverse APIs, we remove evaluation biases and isolate model capabilities from implementation quirks of individual SDKs.

4) Local function APIs are the most common kind of API in typical codebases and studied in many works, e.g., [10, 11, 50]. Invoking a function requires specifying the function name and one argument list. In contrast, general web API invocations are subject to more complex syntax and semantics than such local function calls, as already mentioned in Section 1 (*Scope*).

5) There is a large body of work concerned with enabling LLM-based AI agents to use *tools* to interact with their environment [49]. These tools range from simple utilities like calculators [37] to web services that enable AI agents to perform tasks such as weather queries or restaurant booking [22, 32, 40]. At first sight, this body of work seems related to research on automatic generation of integration code, as tools typically expose a function-like interface. However, both input and output modalities differ between these settings, as well as their non-functional requirements, so benchmarks are not directly transferable to other kinds of API invocations.

API Invocation Datasets. Since functional abstractions are essential in programming, virtually every code dataset includes (function) API invocations. Code datasets involving *web* APIs can be found in the fields of natural language interfaces to web APIs [18, 42], SDK-based API invocations [20, 38], and tool-using AI agents [22, 28, 32, 40, 47]. We are not aware of datasets specifically about generating API integration code to invoke REST APIs like we do.

Evaluation Methods for Web API Invocations. The correctness of AI-generated code is traditionally measured either using exact matches or syntax-based similarity, which both correlate weakly with actual correctness [12], or using functional testing [3], which is not directly transferable to web API invocations, as discussed in Section 2.3. This is why we developed a novel approach described in Section 2. Evaluation methods in related works on API invocations mostly rely on variations of the aforementioned standard approaches. Some works use humans [7] or LLMs [32] as judges, and others execute the API invocation and verify the state of the environment [38, 47]. We are not aware of an evaluation method able to provide results as fine-grained as ours.

[12] https://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part1-protocol.html

Studies on Memorization and Hallucination in Code LLMs. LLMs display a remarkable capacity to *memorize* information from their training data—which can be beneficial when it comes to generating factual responses, but also a problem when it comes to leaking private information from the training data. Studies have shown that LLMs memorize, among other things, method definitions and method calls [48], but also that they struggle to suggest the correct APIs [51]. These findings are in line with our observation that LLMs seem to know many endpoint URLs and argument names, but are often unable to assemble them to a specification-compliant API invocation that solves the task in question.

If LLMs are unable to recall memorized facts, they tend to *hallucinate* information that is not grounded in reality. This is a common error of LLMs, and studies have shown that it also exists in code generation, where, among other things, models may hallucinate function names and arguments [8, 25, 44]. Our study confirms that these findings also hold for web API endpoints and their arguments, where a high amount of hallucinations can be observed as well.

Symbolic Approaches to Detecting API Misuse. Correct usage of APIs is challenging also for humans, as indicated by several studies on API usability issues [33, 35] and real-world API misuses [1, 24, 45]. Several (static analysis) methods have been proposed for detecting (and mitigating) API misuses, e.g., [1, 2, 45]. They are, however, primarily designed to aid humans, not LLMs. Using static analyses for evaluation purposes is generally conceivable, but they can only detect whether an API invocation is legal or illegal, not whether it also solves the given task.

5 Conclusion and Future Work

In this paper, we examined the challenges faced by large language models when generating reliable web API invocation code. We introduced a novel dataset and evaluation pipeline to systematically assess LLM performance in this domain. Our experiments with open-source LLMs revealed significant limitations: Only 30% (full completion) and 40% (argument completion) of the produced samples were completely right. Endpoints and arguments are often hallucinated due to insufficient recall of knowledge on API usage.

Overall, our work underscores the need for integrating quality assurance techniques into AI-driven code generation workflows to ensure correctness and reliability. Our benchmark lays the foundation for future work to investigate the impact of different approaches—such as retrieval-augmented generation [21], fine-tuning, reasoning, feedback loops, and constrained decoding [6, 17]—on the correctness and specification-compliance of generated web API invocation code. While our evaluation pipeline is specialized on JavaScript and Axios, our dataset and methodology is generic and can be transferred to other languages and libraries.

Acknowledgments

Funded by the Hessian Ministry of Higher Education, Research, Science and the Arts within the cluster project *The Third Wave of Artificial Intelligence* (3AI).

References

[1] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: web API misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 507–517. doi:10.1145/2635868.2635916

[2] Simon Binder, Krishna Narasimhan, Svenja Kernig, and Mira Mezini. 2022. jGuard: Programming Misuse-Resilient APIs. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*, Bernd Fischer, Lola Burgueño, and Walter Cazzola (Eds.). ACM, 161–174. doi:10.1145/3567512.3567526

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). https://arxiv.org/abs/2107.03374 arXiv: 2107.03374.

[4] Zheyuan Kevin Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, et al. 2024. The effects of generative AI on high skilled work: Evidence from three field experiments with software developers. *Available at SSRN 4945566* (2024).

[5] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *CoRR* abs/2406.11931 (2024). doi:10.48550/ARXIV.2406.11931 arXiv: 2406.11931.

[6] Daniel Deutsch, Shyam Upadhyay, and Dan Roth. 2019. A General-Purpose Algorithm for Constrained Sequential Inference. In *Proceedings of the 23rd Conference on Computational Natural Language Learning, CoNLL 2019, Hong Kong, China, November 3-4, 2019*, Mohit Bansal and Aline Villavicencio (Eds.). Association for Computational Linguistics, 482–492. doi:10.18653/V1/K19-1045

[7] Hanxing Ding, Shuchang Tao, Liang Pang, Zihao Wei, Jinyang Gao, et al. 2025. ToolCoder: A Systematic Code-Empowered Tool Learning Framework for Large Language Models. *CoRR* abs/2502.11404 (2025). doi:10.48550/ARXIV.2502.11404 arXiv: 2502.11404.

[8] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, et al. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. *CoRR* abs/2407.06153 (2024). doi:10.48550/ARXIV.2407.06153 arXiv: 2407.06153.

[9] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, et al. 2024. The Llama 3 Herd of Models. *CoRR* abs/2407.21783 (2024). doi:10.48550/ARXIV.2407.21783 arXiv: 2407.21783.

[10] Sujan Dutta, Sayantan Mahinder, Raviteja Anantha, and Bortik Bandyopadhyay. 2024. Applying RLAIF for Code Generation with API-usage in Lightweight LLMs. *CoRR* abs/2406.20060 (2024). doi:10.48550/ARXIV.2406.20060 arXiv: 2406.20060.

[11] Aryaz Eghbali and Michael Pradel. 2024. De-Hallucinator: Iterative Grounding for LLM-Based Code Completion. *CoRR* abs/2401.01701 (2024). doi:10.48550/ARXIV.2401.01701 arXiv: 2401.01701.

[12] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the BLEU: How should we assess quality of the Code Generation models? *J. Syst. Softw.* 203 (2023), 111741. doi:10.1016/J.JSS.2023.111741

[13] GitClear 2025. *AI Copilot Code Quality: Evaluating 2024's Increased Defect Rate with Data.* GitClear. Retrieved 2025-05-17 from https://www.gitclear.com/ai_assistant_code_quality_2025_research

[14] Sacha Greif and Eric Burel. 2022. *State of JavaScript 2022.* Devographics. Retrieved 2024-07-30 from https://2022.stateofjs.com/en-US/

[15] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR* abs/2401.14196 (2024). doi:10.48550/ARXIV.2401.14196 arXiv: 2401.14196.

[16] Lianghong Guo, Yanlin Wang, Ensheng Shi, Wanjun Zhong, Hongyu Zhang, et al. 2024. When to Stop? Towards Efficient Code Generation in LLMs with Excess Token Prevention. *CoRR* abs/2407.20042 (2024). doi:10.48550/ARXIV.2407.20042 arXiv: 2407.20042.

[17] Chris Hokamp and Qun Liu. 2017. Lexically Constrained Decoding for Sequence Generation Using Grid Beam Search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 1535–1546. doi:10.18653/v1/P17-1141

[18] Saghar Hosseini, Ahmed Hassan Awadallah, and Yu Su. 2021. Compositional Generalization for Natural Language Interfaces to Web APIs. *CoRR* abs/2112.05209 (2021). https://arxiv.org/abs/2112.05209 arXiv: 2112.05209.

[19] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, et al. 2024. Qwen2.5-Coder Technical Report. *CoRR* abs/2409.12186 (2024). doi:10.48550/ARXIV.2409.12186 arXiv: 2409.12186.

[20] Nihal Jain, Robert Kwiatkowski, Baishakhi Ray, Murali Krishna Ramanathan, and Varun Kumar. 2024. On Mitigating Code LLM Hallucinations with API Documentation. *CoRR* abs/2407.09726 (2024). doi:10.48550/ARXIV.2407.09726 arXiv: 2407.09726.

[21] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, et al. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[22] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, et al. 2023. API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 3102–3116. doi:10.18653/V1/2023.EMNLP-MAIN.187

[23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, et al. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023). doi:10.48550/arXiv.2305.06161 arXiv: 2305.06161.

[24] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. 2021. A Large-scale Study on API Misuses in the Wild. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021.* IEEE, 241–252. doi:10.1109/ICST49551.2021.00034

[25] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, et al. 2024. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. *CoRR* abs/2404.00971 (2024). doi:10.48550/ARXIV.2404.00971 arXiv: 2404.00971.

[26] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *CoRR* abs/2402.19173 (2024). doi:10.48550/ARXIV.2402.19173 arXiv: 2402.19173.

[27] MarketsandMarkets Research Private Ltd. 2024. *API Management Market by Platform, Service – Global Forecast to 2029.* MarketsandMarkets Research Private Ltd. Retrieved 2025-05-18 from https://www.marketsandmarkets.com/Market-Reports/api-management-market-178266736.html

[28] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected with Massive APIs. *CoRR* abs/2305.15334 (2023). doi:10.48550/arXiv.2305.15334 arXiv: 2305.15334.

[29] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022.* IEEE, 754–768. doi:10.1109/SP46214.2022.9833571

[30] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2785–2799. doi:10.1145/3576915.3623157

[31] Postman, Inc. 2024. *2024 State of the API Report.* Postman, Inc. Retrieved 2025-05-17 from https://www.postman.com/state-of-api/2024/

[32] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, et al. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024.* OpenReview.net. https://openreview.net/forum?id=dHng2O0Jjr

[33] Irum Rauf, Elena Troubitsyna, and Ivan Porres. 2019. A systematic mapping study of API usability evaluation methods. *Comput. Sci. Rev.* 33 (2019), 49–68. doi:10.1016/J.COSREV.2019.05.001

[34] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. In *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama Japan, May 8-13, 2021, Extended Abstracts*, Yoshifumi Kitamura, Aaron Quigley, Katherine Isbister, and Takeo Igarashi (Eds.). ACM, 314:1–314:7. doi:10.1145/3411763.3451760

[35] Martin P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Softw.* 26, 6 (2009), 27–34. doi:10.1109/MS.2009.193

[36] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, et al. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). doi:10.48550/ARXIV.2308.12950 arXiv: 2308.12950.

[37] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, et al. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. *CoRR* abs/2302.04761 (2023). doi:10.48550/arXiv.2302.04761 arXiv: 2302.04761.

[38] Raphael Shu, Elman Mansimov, Tamer Alkhouli, Nikolaos Pappas, Salvatore Romeo, et al. 2022. Dialog2API: Task-Oriented Dialogue with API Description and Example Programs. *CoRR* abs/2212.09946 (2022). doi:10.48550/ARXIV.2212.09946 arXiv: 2212.09946.

[39] SmartBear Software 2020. *The State of API 2020 Report.* SmartBear Software. Retrieved 2024-07-29 from https://smartbear.com/resources/ebooks/the-state-of-

api-2020-report/

[40] Yifan Song, Weimin Xiong, Dawei Zhu, Cheng Li, Ke Wang, et al. 2023. RestGPT: Connecting Large Language Models with Real-World Applications via RESTful APIs. *CoRR* abs/2306.06624 (2023). doi:10.48550/ARXIV.2306.06624 arXiv: 2306.06624.

[41] Statista, Inc. 2024. *Most used programming languages among developers worldwide as of 2024*. Statista, Inc. Retrieved 2024-09-08 from https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

[42] Yu Su, Ahmed Hassan Awadallah, Madian Khabsa, Patrick Pantel, Michael Gamon, et al. 2017. Building Natural Language Interfaces to Web APIs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, Ee-Peng Lim, Marianne Winslett, Mark Sanderson, Ada Wai-Chee Fu, Jimeng Sun, J. Shane Culpepper, Eric Lo, Joyce C. Ho, Debora Donato, Rakesh Agrawal, Yu Zheng, Carlos Castillo, Aixin Sun, Vincent S. Tseng, and Chenliang Li (Eds.). ACM, 177–186. doi:10.1145/3132847.3133009

[43] Yu Su, Ahmed Hassan Awadallah, Miaosen Wang, and Ryen W. White. 2018. Natural Language Interfaces with Fine-Grained User Interaction: A Case Study on Web APIs. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval, SIGIR 2018, Ann Arbor, MI, USA, July 08-12, 2018*, Kevyn Collins-Thompson, Qiaozhu Mei, Brian D. Davison, Yiqun Liu, and Emine Yilmaz (Eds.). ACM, 855–864. doi:10.1145/3209978.3210013

[44] Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, and others Desmarais. 2025. Bugs in large language models generated code: an empirical study. *Empir. Softw. Eng.* 30, 3 (2025), 65. doi:10.1007/S10664-025-10614-4

[45] Xiaoke Wang and Lei Zhao. 2023. APICAD: Augmenting API Misuse Detection through Specifications from Code and Documents. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 245–256. doi:10.1109/ICSE48619.2023.00032

[46] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, et al. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *CoRR* abs/2305.07922 (2023). doi:10.48550/arXiv.2305.07922 arXiv: 2305.07922.

[47] Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, et al. 2024. *Berkeley Function Calling Leaderboard*. Retrieved 2025-05-30 from https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html

[48] Zhou Yang, Zhipeng Zhao, Chenyu Wang, Jieke Shi, et al. 2024. Unveiling Memorization in Code Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–13. doi:10.1145/3597503.3639074

[49] Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, et al. 2025. Survey on Evaluation of LLM-based Agents. *CoRR* abs/2503.16416 (2025). doi:10.48550/ARXIV.2503.16416 arXiv: 2503.16416.

[50] Daoguang Zan, Bei Chen, Yongshun Gong, Junzhi Cao, Fengji Zhang, et al. 2023. Private-Library-Oriented Code Generation with Large Language Models. *CoRR* abs/2307.15370 (2023). doi:10.48550/ARXIV.2307.15370 arXiv: 2307.15370.

[51] Terry Yue Zhuo, Xiaoning Du, Zhenchang Xing, Jiamou Sun, Haowei Quan, et al. 2023. Pop Quiz! Do Pre-trained Code Models Possess Knowledge of Correct API Names? *CoRR* abs/2309.07804 (2023). doi:10.48550/ARXIV.2309.07804 arXiv: 2309.07804.

A Data Availability

All source code and data referred to in this paper is available in our artifact[13]. Upon acceptance of the paper, the entire project will be published under an open-source license.

B OpenAPI

OpenAPI provides a structured, human- and machine-readable way to document an API's endpoints, requests, response formats, authentication methods, and other details. For this paper, we define the term *endpoint* as a unique combination of path and HTTP method. Specification files are written in JSON or YAML. Listing 3a shows an excerpt of the OpenAPI specification of the Google Calendar API[14]. Besides some meta information, it contains a list of paths (e.g., `/calendars`) and, for each path, a list of supported HTTP methods (e.g., `post`). Each endpoint has properties for documentation purposes and a list of named parameters (`prettyPrint`). Parameters can be passed in different locations, indicated by the property `in`: `path` parameters are inserted directly into of the URL, `query` parameters are appended to the URL as key–value pairs, and `header` parameters become part of the HTTP request header. Additionally, methods like POST use the `requestBody` to transfer data to the server (e.g., `summary`, `timeZone`). We consider this data as another kind of parameter. Parameters have a `schema` that specifies their data type and constraints on permissible values. The `security` property determines available authentication schemes, such as OAuth 2.0.

Listing 3b shows how a request to the endpoint from the specification in Listing 3a could be implemented in JavaScript using the Axios library. The method called on the `axios` object determines the HTTP method, and the first argument determines the server URL and path. This is followed by one object containing the request body and another object containing header and query parameters (`headers`, `params`). Query parameters are automatically serialized and appended to the URL by Axios. If we execute the code in Listing 3b, a configuration object as shown in Listing 3c is created and a request, configured accordingly, is sent to the given URL. The configuration contains all arguments explicitly given in the code, as well as some implicit parameters, such as `Accept` (for the expected response media type) and `Content-Type` (for the request body media type). The representation of requests as configuration objects is key to our evaluation method described in the paper.

C Technologies

We implemented our evaluation pipeline using the following technologies:

- OpenAPI
- OpenAPI 3 parser
- Axios
- axios-mock-adapter
- Hugging Face Transformers

[13]https://zenodo.org/doi/10.5281/zenodo.13758414 (Note that the artifact also includes work-in-progress implementations for future work, which can be ignored.)
[14]Adapted from https://api.apis.guru/v2/specs/googleapis.com/calendar/v3/openapi.yaml; the original comprises 3190 lines.

D Models

These are the exact names of the models we evaluated:

- `bigcode/starcoderbase`
- `bigcode/starcoder2-3b`
- `bigcode/starcoder2-7b`
- `bigcode/starcoder2-15b`
- `deepseek-ai/deepseek-coder-1.3b-base`
- `deepseek-ai/deepseek-coder-6.7b-base`
- `deepseek-ai/deepseek-coder-7b-base-v1.5`
- `deepseek-ai/deepseek-coder-33b-base`
- `deepseek-ai/DeepSeek-Coder-V2-Lite-Base`
- `google/gemini-pro-1.5`
- `meta-llama/CodeLlama-7b-hf`
- `meta-llama/CodeLlama-13b-hf`
- `meta-llama/CodeLlama-70b-hf`
- `meta-llama/Llama-3.1-8B`
- `meta-llama/Llama-3.1-70B`
- `openai/gpt-4o`
- `openai/gpt-4o-mini`
- `Qwen/Qwen2.5-Coder-0.5B`
- `Qwen/Qwen2.5-Coder-1.5B`
- `Qwen/Qwen2.5-Coder-3B`
- `Qwen/Qwen2.5-Coder-7B`
- `Qwen/Qwen2.5-Coder-14B`
- `Qwen/Qwen2.5-Coder-32B`
- `Salesforce/codet5p-16b`

As can be seen, we use only base (i.e., non-instruction-tuned) models. While instruction-tuned models tend to perform better on coding tasks[15], we considered them to be inappropriate for our setting, which is based on *code completion*. Performing code completion on instruction-tuned models leads to rather unnatural results, as the models often do not directly return the completion and instead start with some response ("Here is your completed code ...") followed by a code snipped that may contain (1) only the completion, (2) the starter code followed by the completion, or (3) an arbitrarily modified version of the starter code and corresponding completion. These factors make the model output very hard to parse reliably. Therefore, we decided to focus our evaluation on base models.

The following hyperparameters were used:

- floating point precision = 16 bit
- # beams = 1
- temperature = 0.0

E Prompts

Listing 4 shows the prompt used for creating the dataset with Gemini 1.5 pro. To avoid incomplete responses when generating the dataset based on OpenAPI specifications with more than 100 endpoints (Asana and Slack), we prompted it multiple times, asking for different subsets of endpoints. The Slack API required additional instructions to account for changes in the API that are not reflected in the OpenAPI specification[16]. Listing 5 shows the prompt used in our evaluation pipeline when generating API invocation code.

[15]Cf., e.g., the EvalPlus leaderboard (https://evalplus.github.io/leaderboard.html)
[16]https://api.slack.com/changelog/2017-10-keeping-up-with-the-jsons

```yaml
openapi: 3.0.0
servers:
  - url: https://www.googleapis.com/calendar/v3
info:
  title: Calendar API
  description: Manipulates events and other calendar data.
paths:
  /calendars:
    post:
      description: Creates a secondary calendar.
      parameters:
        - name: prettyPrint
          description: Returns response with indentations and line breaks.
          in: query
          required: false
          schema:
            type: boolean
      requestBody:
        content:
          application/json:
            schema:
              properties:
                summary:
                  description: Title of the calendar.
                  type: string
                timeZone:
                  description: The time zone of the calendar. (Formatted as an IANA Time Zone Database name, e.g. "Europe/Zurich".) Optional
                      .
                  type: string
              type: object
      security:
        - Oauth2:
            - https://www.googleapis.com/auth/calendar
```

(a) OpenAPI specification (YAML)

```javascript
// Create a secondary calendar with summary "Example Calendar" and
     time zone "America/Los_Angeles". Pretty print the response.
const axios = require('axios');

axios.post('https://www.googleapis.com/calendar/v3/calendars', {
  summary: 'Example Calendar',
  timeZone: 'America/Los_Angeles',
}, {
  headers: {
    Authorization: 'Bearer <access_token>'
  },
  params: {
    prettyPrint: true,
  }
}).then(response => {
  console.log('Calendar created', response.data);
});
```

(b) API invocation (JavaScript)

```json
{
  "headers": {
    "Accept": "application/json, text/plain, */*",
    "Content-Type": "application/json",
    "Authorization": "Bearer <access_token>"
  },
  "params": {
    "prettyPrint": true
  },
  "method": "post",
  "url": "https://www.googleapis.com/calendar/v3/calendars",
  "data": {
    "summary": "Example Calendar",
    "timeZone": "America/Los_Angeles"
  }
}
```

(c) Configuration object (JSON)

Figure 3: Example of a web API and its usage. Top: Excerpt from the Google Calendar OpenAPI specification. Left: JavaScript code to send a request to this API using the Axios library. Right: Configuration object that describes the request sent. For our evaluation, we pair the task description (comment in the JavaScript code) with the configuration to create an input–output sample.

In both cases, we decided against few-shot prompting to increase generalizability and to avoid associated pitfalls. Few-shot prompting techniques mainly help the model to understand the task it is supposed to solve and the desired response format. This is not the issue in the type of task we investigate. Rather, the main problem is that models select the wrong API endpoints and/or pass the wrong arguments to the endpoint. On the other hand, using few-shot prompts introduces new challenges, such as a high sensitivity to the examples provided while reducing the generalizability of the prompt and not guaranteeing inherently superior performance to that of zero-shot prompting [34].

Further optimization of model performance via prompt engineering would require either revealing information about the API's endpoints and parameters—undermining the purpose of our evaluation of the models' memorized knowledge—or tailoring prompts to each individual model (e.g., adjusting wording, formatting, etc.). However, this approach relies on trial-and-error and introduces high uncertainty and variance.

F Extended Results

The full set of metrics calculated by our pipeline is explained in Table 5. Tables 6 and 7 provide a comprehensive summary of these metrics for all evaluated models (cf. Appendix D). Additionally,

Listing 4: Prompt for generating the test data for a given API. Words in curly braces are placeholders.

~~~
Consider this OpenAPI specification:

```yaml
{spec}
~~~

I want you to generate test data for this API. The data should be
      in JSON format and look like this:

```json
{
  "samples": [
    {
      "task": "...",
      "config": {
        "url": "...",
        "method": "...",
        "headers": { ... },
        "params": { ... },
        "data": { ... }
      }
    },
    ...
  ]
}
```

* `samples` is an array containing all the test cases.
* `task` is a natural language description of a specific task that
       can be solved by sending a request to the given API. All
      information required to unambiguously identify and implement
      the corresponding API request must be contained in the task
      description. Therefore, state all expected argument values
      explicitly. The only exception is authentication keys and
      tokens, which must not be specified here (but you may specify
       the authentication method to be used if multiple ones are
      available).
* `config` is an object containing the expected configuration of
      the described request. If a property of `config` is empty, it
       can be omitted.
* `url` is the full URL of the endpoint (server URL + path) and
      may include path parameters but no query parameters.
* `method` is the HTTP method used. It can be one of `get`, `put`,
       `post`, `delete`, and `patch`.
* `headers` is an object containing all expected header arguments.
       Note that the property `"Accept": "application/json, text/
      plain, */*"` is always present and if `data` is sent in the
      request body, `"Content-Type": "mime/type"` is present as
      well (substitute "mime/type" for the respective media type).
* `params` is an object containing all expected query arguments.
* `data` is an object containing all data from the request body.
      It is only used for the methods put, post, delete, and patch.

Remember that for authentication, an additional argument might be
      required. Take a look at an endpoint's `security` property
      and the `securitySchemes` section in the specification, to
      find out if and how to authenticate. I assume you know how
      the usual authentication schemes `apiKey`, `http`, and `
      oauth2` work. Use `<key>` as a placeholder for API keys (e.g
      ., `"name": "<key>"`) and `<token>` as a placeholder for
      authorization tokens (e.g., `"Authorization": "Bearer <token
      >"`).
{api_specific_instructions}
Now, please give me a JSON object that matches my description and
      that contains diverse examples of requests to this API.
{path_selection}
```

the results are broken down by API exemplarily for StarCoder2 and GPT-4o in Tables 8 to 11. Note that some metrics can only be calculated either for *full completion* or for *argument completion* and are therefore not shown in all tables. Values marked with *(t)* are ratios relative to the 395 *total samples*, while values marked with *(e)* are relative to the subset of *executable codes*, which varies from experiment to experiment. The complete raw data these results are based on can be found in our artifact (cf. Appendix A).

Listing 5: Prompt for generating API invocations given a task description. Words in curly braces are placeholders.

```
You are an AI programming assistant that helps users write API
      requests. You are given a comment that describes what the
      user wants to achieve and are supposed to implement it using
      the Axios library in JavaScript. For this, write a single
      call to Axios (with syntax `axios.method(url[, config])`)
      that does exactly what was described in the comment.

* Make sure to include all parameters in `config` that are
      required to solve the given task but don't include any
      unnecessary parameters.
* Insert all values directly into the place where they belong,
      rather than using intermediate variables.
* If the API requires some form of authentication, use `<key>` as
      a placeholder for API keys or `<token>` as a placeholder for
      authorization tokens, respectively.
* If a request body requires a media type other than `text/json`,
      explicitly set the `Content-Type` header to the respective
      type, and Axios will automatically serialize the request body
       accordingly.

Your next task is about the {api} API. Complete the following code
       snippet{extra_instructions}:

```javascript
// {task}
const axios = require('axios');

axios.{method}('{url}',
```

Table 5: Evaluation metrics for correctness and specification-compliance of API invocations.

Metric	Description
Executable codes	Generated code is complete and contains no syntax or runtime error
Correct codes	Generated executable code matches the ground-truth configuration
Illegal codes	Generated code contains at least one violation of the API specification
Correct URLs	Generated URL matches the ground-truth URL
Illegal URLs	Generated URL is not defined in the API specification
Correct methods	Generated HTTP method matches the ground-truth HTTP method
Illegal methods	Generated HTTP method is not defined for the generated URL in the API specification
Correct argument names	Generated arguments are correct
Correct argument values	Generated argument values are correct
Missing arguments	Expected arguments are not generated
Unexpected arguments	Generated arguments are not expected
Unnecessary arguments	Redundant arguments are generated for an API endpoint
Illegal arguments	Generated arguments are not permitted for the generated API endpoint
Mean argument precision	Probability that the generated arguments are correct
Mean argument recall	Probability that the correct arguments are generated
Mean argument Jaccard index	Overlap between generated and correct arguments
Mean argument value conditional accuracy	Probability that an argument value is correct if the argument name is correct
Total errors	Any type of error prevented execution
Incomplete codes	Generated code did not contain a complete API invocation
Runtime errors	Generated code produced an error when trying to execute it

Table 6: Complete evaluation results for *full completion*.

	CodeT5+ (16B)	StarCoder (15.5B)	StarCoder2 (3B)	StarCoder2 (7B)	StarCoder2 (15B)	DeepSeek-Coder (1.3B)	DeepSeek-Coder (6.7B)	DeepSeek-Coder (7B)	DeepSeek-Coder (33B)	DeepSeek-Coder-V2 (16B)	Qwen2.5-Coder (0.5B)	Qwen2.5-Coder (1.5B)	Qwen2.5-Coder (3B)	Qwen2.5-Coder (7B)	Qwen2.5-Coder (14B)	Qwen2.5-Coder (32B)	Code Llama (7B)	Code Llama (13B)	Code Llama (70B)	Llama 3.1 (8B)	Llama 3.1 (70B)	Gemini 1.5 Pro	GPT-4o mini	GPT-4o
Executable implementations (t)	0.95	0.94	0.97	0.95	0.99	0.96	0.96	0.57	0.98	0.95	0.01	0.00	0.00	0.00	0.00	0.00	0.99	0.89	0.99	0.00	0.00	**1.00**	**1.00**	**1.00**
Correct implementations (t)	0.06	0.12	0.11	0.10	0.26	0.08	0.18	0.04	0.17	0.21	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.08	0.30	0.00	0.00	0.45	0.39	**0.60**
Correct implementations (e)	0.06	0.13	0.12	0.11	0.27	0.08	0.18	0.06	0.17	0.21	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.09	0.31	0.00	0.00	0.45	0.39	**0.60**
Correct URLs (t)	0.31	0.51	0.40	0.45	0.60	0.33	0.57	0.35	0.67	0.51	0.01	0.00	0.00	0.00	0.00	0.00	0.48	0.51	0.72	0.00	0.00	0.72	0.57	**0.78**
Correct URLs (e)	0.32	0.54	0.41	0.48	0.60	0.35	0.60	0.62	0.68	0.53	0.67	0.00	0.00	0.00	0.00	0.00	0.48	0.58	0.72	0.00	0.00	0.72	0.57	**0.78**
Illegal URLs (t)	0.37	0.17	0.22	0.27	0.22	0.32	0.19	0.10	0.13	0.20	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.28	0.21	0.15	**0.00**	**0.00**	0.19	0.23	0.09
Illegal URLs (e)	0.39	0.18	0.23	0.29	0.22	0.34	0.20	0.17	0.14	0.21	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.28	0.24	0.15	**0.00**	**0.00**	0.20	0.23	0.09
Correct methods (t)	0.81	0.83	0.78	0.88	0.86	0.79	0.84	0.53	0.86	0.81	0.01	0.00	0.00	0.00	0.00	0.00	0.73	0.75	0.88	0.00	0.00	**0.91**	0.83	**0.91**
Correct methods (e)	0.85	0.88	0.81	0.93	0.87	0.83	0.88	0.93	0.87	0.84	**1.00**	0.00	0.00	0.00	0.00	0.00	0.74	0.84	0.89	0.00	0.00	0.92	0.83	0.91
Illegal methods (t)	0.12	0.09	0.12	0.05	0.11	0.12	0.10	0.03	0.10	0.13	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.22	0.13	0.10	**0.00**	**0.00**	0.07	0.14	0.07
Illegal methods (e)	0.13	0.09	0.12	0.05	0.11	0.12	0.11	0.05	0.10	0.14	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.23	0.14	0.10	**0.00**	**0.00**	0.07	0.14	0.07
Correct argument names (t)	0.39	0.50	0.49	0.41	0.63	0.45	0.53	0.25	0.58	0.52	0.00	0.00	0.00	0.00	0.00	0.00	0.37	0.56	0.68	0.00	0.00	0.79	0.76	**0.86**
Correct argument names (e)	0.41	0.54	0.51	0.43	0.63	0.47	0.56	0.48	0.59	0.55	0.29	0.00	0.00	0.00	0.00	0.00	0.38	0.63	0.68	0.00	0.00	0.80	0.76	**0.87**
Correct argument values (t)	0.29	0.43	0.40	0.33	0.58	0.39	0.47	0.18	0.53	0.48	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.39	0.63	0.00	0.00	0.76	0.72	**0.83**
Correct argument values (e)	0.30	0.46	0.42	0.35	0.58	0.41	0.50	0.34	0.54	0.50	0.29	0.00	0.00	0.00	0.00	0.00	0.33	0.43	0.63	0.00	0.00	0.77	0.72	**0.83**
Missing arguments (t)	0.61	0.50	0.51	0.59	0.37	0.55	0.47	0.75	0.42	0.48	1.00	1.00	1.00	1.00	1.00	1.00	0.63	0.44	0.32	1.00	1.00	0.21	0.24	**0.14**
Missing arguments (e)	0.59	0.46	0.49	0.57	0.37	0.53	0.44	0.52	0.41	0.45	0.71	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.62	0.37	0.32	**0.00**	**0.00**	0.20	0.24	0.13
Unexpected arguments (t)	0.30	0.25	0.34	0.31	0.25	0.32	0.25	0.11	0.24	0.22	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.17	0.22	0.20	**0.00**	**0.00**	0.18	0.17	0.11
Unexpected arguments (e)	0.31	0.26	0.35	0.32	0.25	0.33	0.26	0.20	0.25	0.23	0.12	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.17	0.24	0.20	**0.00**	**0.00**	0.18	0.17	0.11
Mean argument precision (t)	0.50	0.61	0.53	0.49	0.67	0.53	0.62	0.36	0.59	0.63	0.01	0.00	0.00	0.00	0.00	0.00	0.56	0.65	0.75	0.00	0.00	0.82	0.82	**0.89**
Mean argument precision (e)	0.52	0.65	0.54	0.51	0.68	0.55	0.65	0.63	0.60	0.67	0.67	0.00	0.00	0.00	0.00	0.00	0.57	0.72	0.75	0.00	0.00	0.82	0.82	**0.89**
Mean argument recall (t)	0.41	0.51	0.48	0.42	0.63	0.47	0.54	0.28	0.56	0.55	0.00	0.00	0.00	0.00	0.00	0.00	0.40	0.58	0.69	0.00	0.00	0.81	0.78	**0.87**
Mean argument recall (e)	0.43	0.54	0.50	0.44	0.63	0.49	0.56	0.49	0.56	0.57	0.33	0.00	0.00	0.00	0.00	0.00	0.41	0.65	0.69	0.00	0.00	0.82	0.78	**0.87**
Mean arg. Jaccard index (t)	0.34	0.46	0.40	0.35	0.56	0.38	0.48	0.25	0.47	0.49	0.00	0.00	0.00	0.00	0.00	0.00	0.37	0.52	0.62	0.00	0.00	0.75	0.72	**0.83**
Mean arg. Jaccard index (e)	0.36	0.48	0.41	0.37	0.56	0.40	0.50	0.44	0.48	0.52	0.33	0.00	0.00	0.00	0.00	0.00	0.37	0.58	0.63	0.00	0.00	0.75	0.72	**0.83**
Mean arg. val. cond. acc. (t)	0.51	0.64	0.60	0.57	0.78	0.66	0.69	0.31	0.74	0.72	0.01	0.00	0.00	0.00	0.00	0.00	0.58	0.51	0.84	0.00	0.00	**0.95**	**0.95**	**0.95**
Mean arg. val. cond. acc. (e)	0.53	0.68	0.62	0.60	0.79	0.69	0.72	0.54	0.75	0.76	0.67	0.00	0.00	0.00	0.00	0.00	0.59	0.57	0.84	0.00	0.00	**0.96**	**0.96**	0.95
Total errors	18	22	13	21	3	16	16	169	6	18	392	395	393	395	395	390	5	42	2	395	395	**1**	**1**	**1**
Incomplete implementations	14	5	11	5	2	12	3	2	6	9	29	35	12	2	160	**0**	4	5	1	4	**0**	1	**0**	**0**
Runtime errors	4	17	2	16	1	4	13	167	**0**	9	363	360	381	393	235	390	1	37	1	391	395	**0**	1	1

Table 7: Complete evaluation results for *argument completion*.

	CodeT5+ (16B)	StarCoder (15.5B)	StarCoder2 (3B)	StarCoder2 (7B)	StarCoder2 (15B)	DeepSeek-Coder (1.3B)	DeepSeek-Coder (6.7B)	DeepSeek-Coder (7B)	DeepSeek-Coder (33B)	DeepSeek-Coder-V2 (16B)	Qwen2.5-Coder (0.5B)	Qwen2.5-Coder (1.5B)	Qwen2.5-Coder (3B)	Qwen2.5-Coder (7B)	Qwen2.5-Coder (14B)	Qwen2.5-Coder (32B)	Code Llama (7B)	Code Llama (13B)	Code Llama (70B)	Llama 3.1 (8B)	Llama 3.1 (70B)	Gemini 1.5 Pro	GPT-4o mini	GPT-4o
Executable implementations (t)	0.94	0.97	0.98	0.99	0.99	0.97	0.99	0.99	0.99	0.98	0.89	0.95	0.92	0.99	0.93	0.99	0.99	0.95	**1.00**	0.98	0.99	**1.00**	0.99	**1.00**
Correct implementations (t)	0.12	0.27	0.21	0.19	0.25	0.21	0.26	0.08	0.25	0.35	0.08	0.03	0.16	0.28	0.37	0.38	0.23	0.12	0.40	0.05	0.29	0.61	0.63	**0.77**
Correct implementations (e)	0.13	0.27	0.22	0.19	0.25	0.22	0.27	0.08	0.25	0.36	0.09	0.03	0.18	0.28	0.40	0.38	0.23	0.13	0.40	0.05	0.29	0.61	0.64	**0.77**
Illegal implementations (t)	0.59	0.27	0.48	0.47	0.35	0.51	0.33	0.21	0.38	0.31	0.58	0.50	0.44	0.41	0.32	0.32	0.11	0.28	0.23	0.30	0.26	0.17	0.20	**0.09**
Illegal implementations (e)	0.63	0.27	0.49	0.47	0.36	0.52	0.33	0.22	0.39	0.32	0.65	0.53	0.48	0.41	0.35	0.32	0.11	0.30	0.23	0.31	0.26	0.17	0.20	**0.09**
Correct argument names (t)	0.49	0.71	0.62	0.53	0.64	0.60	0.66	0.52	0.69	0.67	0.43	0.47	0.44	0.67	0.65	0.80	0.48	0.70	0.78	0.43	0.63	0.90	0.88	**0.93**
Correct argument names (e)	0.52	0.73	0.63	0.54	0.65	0.62	0.67	0.53	0.69	0.69	0.49	0.50	0.48	0.67	0.70	0.81	0.48	0.73	0.79	0.44	0.64	0.90	0.89	**0.93**
Correct argument values (t)	0.42	0.61	0.52	0.44	0.57	0.52	0.60	0.45	0.64	0.62	0.34	0.41	0.40	0.63	0.58	0.73	0.42	0.51	0.72	0.38	0.58	0.85	0.84	**0.90**
Correct argument values (e)	0.45	0.62	0.53	0.44	0.58	0.54	0.61	0.46	0.65	0.63	0.40	0.43	0.43	0.63	0.63	0.74	0.42	0.53	0.72	0.39	0.58	0.85	0.84	**0.90**
Missing arguments (t)	0.51	0.29	0.38	0.47	0.36	0.40	0.34	0.48	0.31	0.33	0.57	0.53	0.56	0.33	0.35	0.20	0.52	0.30	0.22	0.57	0.37	0.10	0.12	**0.07**
Missing arguments (e)	0.48	0.27	0.37	0.46	0.35	0.38	0.33	0.47	0.31	0.31	0.51	0.50	0.52	0.33	0.30	0.19	0.52	0.27	0.21	0.56	0.36	0.10	0.11	**0.07**
Unnecessary arguments (t)	0.03	0.05	0.08	0.07	0.07	0.05	0.07	0.06	0.07	0.03	0.03	0.03	0.03	0.06	0.03	0.07	0.05	0.05	0.05	0.03	0.04	0.04	0.02	**0.01**
Unnecessary arguments (e)	0.03	0.05	0.08	0.07	0.07	0.05	0.07	0.06	0.07	0.03	0.03	0.03	0.03	0.06	0.03	0.07	0.05	0.05	0.05	0.03	0.04	0.04	0.02	**0.01**
Illegal arguments (t)	0.24	0.13	0.21	0.20	0.14	0.23	0.15	0.08	0.13	0.13	0.29	0.24	0.23	0.17	0.18	0.11	0.06	0.13	0.09	0.16	0.12	0.09	0.09	**0.05**
Illegal arguments (e)	0.25	0.13	0.21	0.20	0.14	0.24	0.15	0.09	0.13	0.13	0.31	0.25	0.24	0.17	0.20	0.11	0.06	0.13	0.09	0.16	0.12	0.09	0.09	**0.05**
Mean argument precision (t)	0.55	0.79	0.60	0.61	0.71	0.64	0.73	0.73	0.69	0.76	0.48	0.59	0.58	0.71	0.69	0.78	0.66	0.75	0.83	0.63	0.78	0.88	0.89	**0.93**
Mean argument precision (e)	0.58	0.81	0.62	0.61	0.71	0.65	0.74	0.74	0.69	0.77	0.54	0.62	0.63	0.71	0.74	0.78	0.66	0.79	0.83	0.65	0.79	0.88	0.90	**0.94**
Mean argument recall (t)	0.50	0.72	0.61	0.53	0.64	0.63	0.66	0.52	0.66	0.70	0.46	0.46	0.46	0.68	0.67	0.78	0.53	0.70	0.78	0.42	0.66	0.90	0.90	**0.93**
Mean argument recall (e)	0.53	0.74	0.62	0.54	0.65	0.64	0.67	0.52	0.67	0.71	0.52	0.49	0.50	0.68	0.72	0.79	0.53	0.74	0.79	0.43	0.67	0.90	0.90	**0.94**
Mean arg. Jaccard index (t)	0.42	0.66	0.51	0.46	0.57	0.54	0.59	0.46	0.57	0.64	0.38	0.39	0.41	0.59	0.62	0.71	0.49	0.64	0.72	0.38	0.61	0.85	0.85	**0.91**
Mean arg. Jaccard index (e)	0.45	0.68	0.52	0.47	0.57	0.55	0.60	0.47	0.57	0.65	0.43	0.41	0.45	0.59	0.67	0.71	0.49	0.67	0.73	0.39	0.61	0.85	0.86	**0.92**
Mean arg. val. cond. acc. (t)	0.66	0.80	0.68	0.71	0.82	0.75	0.82	0.77	0.84	0.84	0.59	0.71	0.68	0.88	0.72	0.87	0.65	0.62	0.89	0.66	0.83	0.94	0.94	**0.95**
Mean arg. val. cond. acc. (e)	0.71	0.83	0.70	0.72	0.83	0.77	0.83	0.78	0.84	0.86	0.67	0.74	0.74	0.89	0.78	0.87	0.65	0.65	0.89	0.67	0.84	0.94	**0.95**	**0.95**
Total errors	23	11	9	4	3	11	4	5	3	6	45	19	32	2	28	2	3	19	**1**	9	5	**1**	2	**1**
Incomplete implementations	22	5	7	4	1	11	4	3	3	6	44	17	31	2	27	2	3	5	1	6	1	1	**0**	**0**
Runtime errors	1	6	2	**0**	2	**0**	**0**	2	**0**	**0**	1	2	1	**0**	1	**0**	**0**	14	**0**	3	4	**0**	2	1

Table 8: Evaluation results of StarCoder2 by API for *full completion.*

	Overall	Asana	Google Calendar	Google Sheets	Slack
Executable implementations (t)	0.99	0.99	**1.00**	**1.00**	0.99
Correct implementations (t)	0.26	0.37	**0.46**	0.18	0.13
Correct implementations (e)	0.27	0.37	**0.46**	0.18	0.13
Correct URLs (t)	0.60	0.60	**0.89**	0.47	0.54
Correct URLs (e)	0.60	0.61	**0.89**	0.47	0.55
Illegal URLs (t)	0.22	0.25	**0.03**	0.18	0.22
Illegal URLs (e)	0.22	0.25	**0.03**	0.18	0.23
Correct methods (t)	0.86	**0.90**	0.89	0.47	0.86
Correct methods (e)	0.87	**0.90**	0.89	0.47	0.87
Illegal methods (t)	0.11	0.10	**0.05**	0.29	0.11
Illegal methods (e)	0.11	0.10	**0.05**	0.29	0.11
Correct argument names (t)	0.63	0.73	**0.81**	0.64	0.49
Correct argument names (e)	0.63	0.74	**0.81**	0.64	0.49
Correct argument values (t)	0.58	0.64	**0.79**	0.59	0.46
Correct argument values (e)	0.58	0.64	**0.79**	0.59	0.47
Missing arguments (t)	0.37	0.27	**0.19**	0.36	0.51
Missing arguments (e)	0.37	0.26	**0.19**	0.36	0.51
Unexpected arguments (t)	0.25	**0.11**	0.18	0.26	0.36
Unexpected arguments (e)	0.25	**0.11**	0.18	0.26	0.37
Mean argument precision (t)	0.67	**0.88**	0.76	0.60	0.46
Mean argument precision (e)	0.68	**0.88**	0.76	0.60	0.47
Mean argument recall (t)	0.63	**0.76**	0.75	0.66	0.47
Mean argument recall (e)	0.63	**0.76**	0.75	0.66	0.48
Mean arg. Jaccard index (t)	0.56	**0.71**	0.68	0.52	0.38
Mean arg. Jaccard index (e)	0.56	**0.72**	0.68	0.52	0.39
Mean arg. val. cond. acc. (t)	0.78	0.85	**0.87**	0.84	0.70
Mean arg. val. cond. acc. (e)	0.79	0.85	**0.87**	0.84	0.70
Total errors	3	1	**0**	**0**	2
Incomplete implementations	2	1	**0**	**0**	1
Runtime errors	1	**0**	**0**	**0**	1
Timeouts	**0**	**0**	**0**	**0**	**0**
Unsatisfiable constraints	**0**	**0**	**0**	**0**	**0**

Table 9: Evaluation results of StarCoder2 by API for *argument completion.*

	Overall	Asana	Google Calendar	Google Sheets	Slack
Executable implementations (t)	0.99	0.99	**1.00**	**1.00**	0.99
Correct implementations (t)	0.25	0.37	**0.46**	0.24	0.10
Correct implementations (e)	0.25	0.37	**0.46**	0.24	0.10
Illegal implementations (t)	0.35	0.23	**0.08**	0.24	0.55
Illegal implementations (e)	0.36	0.23	**0.08**	0.24	0.55
Correct argument names (t)	0.64	0.69	**0.86**	0.67	0.55
Correct argument names (e)	0.65	0.69	**0.86**	0.67	0.55
Correct argument values (t)	0.57	0.59	**0.81**	0.61	0.49
Correct argument values (e)	0.58	0.59	**0.81**	0.61	0.50
Missing arguments (t)	0.36	0.31	**0.14**	0.33	0.45
Missing arguments (e)	0.35	0.31	**0.14**	0.33	0.45
Unnecessary arguments (t)	0.07	**0.03**	0.10	0.06	0.10
Unnecessary arguments (e)	0.07	**0.03**	0.10	0.06	0.10
Illegal arguments (t)	0.14	0.11	**0.02**	0.06	0.20
Illegal arguments (e)	0.14	0.11	**0.02**	0.06	0.20
Mean argument precision (t)	0.71	**0.84**	0.82	0.83	0.55
Mean argument precision (e)	0.71	**0.85**	0.82	0.83	0.55
Mean argument recall (t)	0.64	0.72	**0.80**	0.68	0.53
Mean argument recall (e)	0.65	0.72	**0.80**	0.68	0.54
Mean arg. Jaccard index (t)	0.57	0.68	**0.74**	0.64	0.42
Mean arg. Jaccard index (e)	0.57	0.69	**0.74**	0.64	0.43
Mean arg. val. cond. acc. (t)	0.82	0.84	**0.91**	0.85	0.78
Mean arg. val. cond. acc. (e)	0.83	0.84	**0.91**	0.85	0.79
Total errors	3	1	**0**	**0**	2
Incomplete implementations	1	**0**	**0**	**0**	1
Runtime errors	2	1	**0**	**0**	1
Timeouts	**0**	**0**	**0**	**0**	**0**
Unsatisfiable constraints	**0**	**0**	**0**	**0**	**0**

Table 10: Evaluation results of GPT-4o by API for *full completion.*

	Overall	Asana	Google Calendar	Google Sheets	Slack
Executable implementations (t)	**1.00**	0.99	**1.00**	**1.00**	**1.00**
Correct implementations (t)	0.60	0.56	**0.89**	0.59	0.57
Correct implementations (e)	0.60	0.57	**0.89**	0.59	0.57
Correct URLs (t)	0.78	0.75	**1.00**	0.76	0.75
Correct URLs (e)	0.78	0.76	**1.00**	0.76	0.75
Illegal URLs (t)	0.09	0.12	**0.00**	**0.00**	0.10
Illegal URLs (e)	0.09	0.12	**0.00**	**0.00**	0.10
Correct methods (t)	0.91	0.90	0.89	0.88	**0.93**
Correct methods (e)	0.91	0.90	0.89	0.88	**0.93**
Illegal methods (t)	0.07	0.10	**0.00**	**0.00**	0.07
Illegal methods (e)	0.07	0.10	**0.00**	**0.00**	0.07
Correct argument names (t)	0.86	0.83	**1.00**	0.88	0.86
Correct argument names (e)	0.87	0.84	**1.00**	0.88	0.86
Correct argument values (t)	0.83	0.79	**1.00**	0.81	0.83
Correct argument values (e)	0.83	0.80	**1.00**	0.81	0.83
Missing arguments (t)	0.14	0.17	**0.00**	0.12	0.14
Missing arguments (e)	0.13	0.16	**0.00**	0.12	0.14
Unexpected arguments (t)	0.11	0.11	**0.00**	0.06	0.13
Unexpected arguments (e)	0.11	0.11	**0.00**	0.06	0.13
Mean argument precision (t)	0.89	0.90	**1.00**	0.93	0.85
Mean argument precision (e)	0.89	0.90	**1.00**	0.93	0.85
Mean argument recall (t)	0.87	0.85	**1.00**	0.89	0.86
Mean argument recall (e)	0.87	0.86	**1.00**	0.89	0.86
Mean arg. Jaccard index (t)	0.83	0.81	**1.00**	0.86	0.81
Mean arg. Jaccard index (e)	0.83	0.81	**1.00**	0.86	0.81
Mean arg. val. cond. acc. (t)	0.95	0.94	**1.00**	0.93	0.95
Mean arg. val. cond. acc. (e)	0.95	0.95	**1.00**	0.93	0.95
Total errors	1	1	**0**	**0**	**0**
Incomplete implementations	**0**	**0**	**0**	**0**	**0**
Runtime errors	1	1	**0**	**0**	**0**
Timeouts	**0**	**0**	**0**	**0**	**0**
Unsatisfiable constraints	**0**	**0**	**0**	**0**	**0**

Table 11: Evaluation results of GPT-4o by API for *argument completion.*

	Overall	Asana	Google Calendar	Google Sheets	Slack
Executable implementations (t)	**1.00**	0.99	**1.00**	**1.00**	**1.00**
Correct implementations (t)	0.77	0.77	**0.86**	0.71	0.75
Correct implementations (e)	0.77	0.77	**0.86**	0.71	0.75
Illegal implementations (t)	0.09	0.07	0.14	**0.06**	0.11
Illegal implementations (e)	0.09	0.07	0.14	**0.06**	0.11
Correct argument names (t)	0.93	**0.96**	0.93	0.92	0.91
Correct argument names (e)	0.93	**0.97**	0.93	0.92	0.91
Correct argument values (t)	0.90	0.91	**0.93**	0.89	0.88
Correct argument values (e)	0.90	0.92	**0.93**	0.89	0.88
Missing arguments (t)	0.07	**0.04**	0.07	0.08	0.09
Missing arguments (e)	0.07	**0.03**	0.07	0.08	0.09
Unnecessary arguments (t)	0.01	0.02	**0.00**	**0.00**	0.02
Unnecessary arguments (e)	0.01	0.02	**0.00**	**0.00**	0.02
Illegal arguments (t)	0.05	**0.02**	0.07	**0.02**	0.07
Illegal arguments (e)	0.05	**0.02**	0.07	**0.02**	0.07
Mean argument precision (t)	0.93	0.96	0.93	**0.98**	0.90
Mean argument precision (e)	0.94	0.97	0.93	**0.98**	0.90
Mean argument recall (t)	0.93	**0.97**	0.93	0.93	0.91
Mean argument recall (e)	0.94	**0.97**	0.93	0.93	0.91
Mean arg. Jaccard index (t)	0.91	**0.95**	0.91	0.92	0.88
Mean arg. Jaccard index (e)	0.92	**0.95**	0.91	0.92	0.88
Mean arg. val. cond. acc. (t)	0.95	0.95	**1.00**	0.96	0.94
Mean arg. val. cond. acc. (e)	0.95	0.96	**1.00**	0.96	0.94
Total errors	1	1	**0**	**0**	**0**
Incomplete implementations	**0**	**0**	**0**	**0**	**0**
Runtime errors	1	1	**0**	**0**	**0**
Timeouts	**0**	**0**	**0**	**0**	**0**
Unsatisfiable constraints	**0**	**0**	**0**	**0**	**0**