On Mitigating Code LLM Hallucinations with API Documentation

Nihal Jain
Amazon Web Services, USA
nihjain@amazon.com

Robert Kwiatkowski*
robert.kwiatkowski@columbia.edu

Baishakhi Ray
Amazon Web Services, USA
rabaisha@amazon.com

Murali Krishna Ramanathan
Amazon Web Services, USA
mkraman@amazon.com

Varun Kumar
Amazon Web Services, USA
kuvrun@amazon.com

***Abstract*—In this study, we address the issue of API hallucinations in various software engineering contexts. We introduce `CloudAPIBench`, a new benchmark designed to measure API hallucination occurrences. `CloudAPIBench` also provides annotations for frequencies of API occurrences in the public domain, allowing us to study API hallucinations at various frequency levels. Our findings reveal that Code LLMs struggle with low frequency APIs: for *e.g.*, GPT-4o achieves only 38.58% valid low frequency API invocations. We demonstrate that Documentation Augmented Generation (DAG) significantly improves performance for low frequency APIs (increase to 47.94% with DAG) but negatively impacts high frequency APIs when using sub-optimal retrievers (a 39.02% absolute drop). To mitigate this, we propose to intelligently trigger DAG where we check against an API index or leverage Code LLMs' confidence scores to retrieve only when needed. We demonstrate that our proposed methods enhance the balance between low and high frequency API performance, resulting in more reliable API invocations (8.20% absolute improvement on `CloudAPIBench` for GPT-4o).**

***Index Terms*—code generation, hallucinations, retrieval augmented generation, API invocations**

I. Introduction

Large Language Models for code generation (Code LLMs) are being increasingly used by developers in industries [2]–[4]. Major software companies like Meta, Google and Amazon are actively adopting AI-powered coding assistants relying on these models to enhance the productivity of their vast software development teams [5]–[7]. As these tools become integral to modern development workflows, ensuring that Code LLMs generate high quality code is essential. However, as noted in [8], these tools often introduce critical errors in code that developers struggle to localize and rectify, making debugging more challenging.

We show that a major mode of failure of Code LLMs is in generating invalid API invocations, *i.e.*, *API hallucinations*. These inaccuracies – where models invoke non-existent APIs, misuse API syntax, or generate fictitious arguments – pose significant challenges for developers. Software teams may rely heavily on third-party API providers, such as Amazon Web Services (AWS) and Microsoft Azure for cloud services. For such developers using Code LLMs, API hallucinations can lead to critical failures in production and increased debugging costs. Hence, in this work, we focus on the occurrence of API hallucinations in the context of cloud-based APIs.

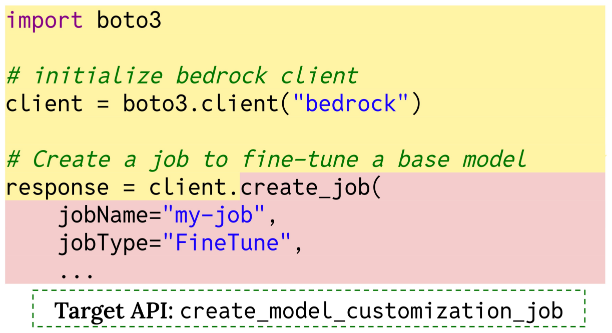
The problem of API hallucinations is exacerbated by the rapid evolution of APIs, including frequent updates and deprecation of existing APIs [9]. Consequently, new, updated, or infrequently used APIs (*i.e.*, *low frequency* APIs) are more prone to hallucinations (see Figure 1 (left)). To systematically measure the prevalence of these errors, we introduce **`CloudAPIBench`**, a benchmark specifically designed to evaluate API hallucinations for major cloud providers like AWS and Microsoft Azure. This makes `CloudAPIBench` a practical tool for measuring API hallucinations in real-world software engineering contexts that rely heavily on cloud services.

Further, we explore mitigation strategies for API hallucinations. When uncertain about API usage, human developers frequently rely on API documentation. Likewise, we hypothesize that Code LLMs should consult these resources under uncertainty. Hence, to address API hallucinations, we adopt retrieval augmented generation with documentation, *i.e.*, **Documentation Augmented Generation (DAG)**, which has shown early promise [10], [11].

However, DAG may be unnecessary when APIs are stable or well-covered in the model's training data (*i.e.*, *high frequency* APIs)—we find that DAG with suboptimal retrievers indeed degrades performance for high frequency APIs, supporting that LLMs are sensitive to irrelevant information [12], [13]. As such, we also present two simple yet effective strategies that can be easily adapted with DAG to address such pitfalls.

Figure 1 (right) demonstrates how the frequency of an API's occurrence in the public domain affects Code LLMs. We analyze the perplexity of StarCoder2-15B (base model) [14] on API tokens across two frequency groups: *low* (≤ 10 occurrences in training data: The Stack v2) and *high* (≥ 100 occurrences), with detailed frequency descriptions in Section II-B. The base model excels with high frequency APIs but struggles with low frequency ones. While DAG enhances performance for low frequency APIs, it compromises high frequency API performance due to occasional irrelevant augmentations from

*Work done while at Amazon



```
import boto3

# initialize bedrock client
client = boto3.client("bedrock")

# Create a job to fine-tune a base model
response = client.create_job(
    jobName="my-job",
    jobType="FineTune",
    ...
```

Target API: `create_model_customization_job`

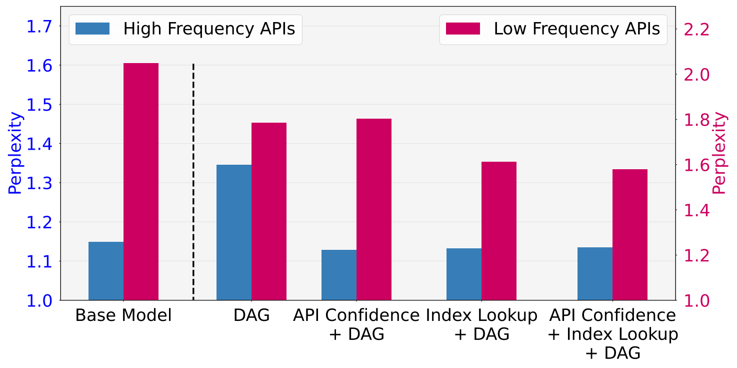


Fig. 1: **Introduction. (Left)** A `CloudAPIBench` task (yellow) and StarCoder2-15B's response (red) are displayed. The target is a recently released AWS API [1], *i.e.*, a *low frequency* API. Due to limited training on such APIs, the Code LLM hallucinates a non-existent API invocation. **(Right)** Given a prompt from `CloudAPIBench`, we measure the perplexity of the target API tokens using StarCoder2-15B (*lower is better*). The base model handles high frequency APIs well but falters with low frequency ones. While DAG (with imperfect retrievers) improves low frequency API performance, it hurts high frequency API performance due to irrelevant augmentations. This paper's methods and analyses address this limitation of DAG.

suboptimal retrievers: these distract the Code LLM's reliance on its internal knowledge, which is sufficient for high frequency APIs. To address DAG's limitations, we explore methods such as inspecting model confidence scores [15], [16] and validating model generations against an API index before retrieval. These strategies effectively mitigate DAG's drawbacks, enhancing the reliability of Code LLMs.

We outline our key contributions and the structure of the paper as follows:

- We introduce `CloudAPIBench` to systematically study real-world API hallucinations; this benchmark evaluates API hallucinations across major cloud SDK APIs, *i.e.*, those by AWS and Azure (**Section II**).
- We present a thorough study of DAG to enhance `CloudAPIBench` performance, identifying the parts of documentation that reduce hallucinations and quantifying the impact of other retrieval components on model efficacy (**Section III**).
- We characterize scenarios where DAG may degrade performance and discuss selective retrieval methods to improve Code LLMs' utilization of documentation (**Section IV**).

We believe this is the first work to measure and characterize real-world API hallucinations for various Code LLMs and explore strategies to mitigate this issue through documentation.

II. API Hallucinations & `CloudAPIBench`

We first comment on the impact of API hallucinations in and then introduce `CloudAPIBench`.

A. Impact of API Hallucinations

Reference [17] identify that API hallucinations constitute up to 15% of all hallucinations in state-of-the-art Code LLMs, influencing cloud software engineering where code is API-intensive. These hallucinations can propagate errors, creating a snowball effect [18]. For *e.g.*, a hallucinated API call can lead to hallucinated handling of its response in subsequent code segments, compounding the problem [19]. As adoption of Code LLMs grows, the cognitive burden on developers increases [8], as they must trace and rectify both the initial hallucination and all affected code segments.

Model	Edit Similarity to GT (%) (↑)	
	Correct API	**Hallucinated API**
StarCoder2-2B	40.29	36.61
StarCoder2-7B	42.05	40.27
StarCoder2-15B	43.85	39.67

TABLE I: **Impact of API hallucinations**. Code LLMs tend to diverge further away from the groundtruth code when supplied with a hallucinated API name in the prompt.

We provide concrete evidence that API hallucinations lead to code failures in Table I. We sample 171 Python programs with AWS API invocations from the Stack v2 and cut the program after the name of the API; the following code is considered as the groundtruth and we prompt Code LLMs to continue the incomplete program. When the name of the API in the incomplete program is replaced with a hallucinated name (*e.g.*, `get_item` → `fetch_item`, *etc.*), the model generates code that diverges further away from the groundtruth as measured by Edit Similarity [20] in Table I, potentially introducing errors that need to be fixed by developers. Given this impact of API hallucinations, it is critical to explore methods for their detection and mitigation, as done in this work.

B. `CloudAPIBench`

Current benchmarks evaluate various programming skills such as problem solving [3], [21], repository-level coding [20], [22], and tool usage [11], [23]. However, a comprehensive benchmark for assessing real-world API hallucinations in software engineering remains absent. To address this, we

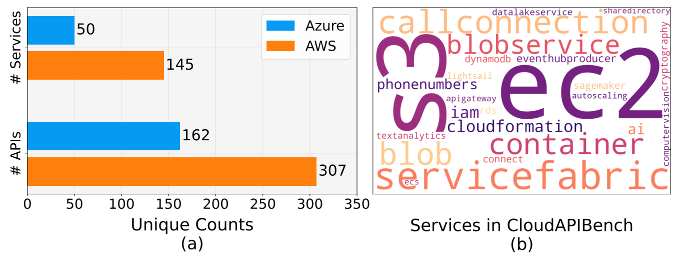


Fig. 2: **Composition of `CloudAPIBench`.** (a) The benchmark comprises diverse APIs from various AWS and Azure services. (b) Word cloud visualizing the services in `CloudAPIBench`; from AWS **`s3`** to Azure **`computervision`**, `CloudAPIBench` comprises many cloud-based software engineering use-cases.

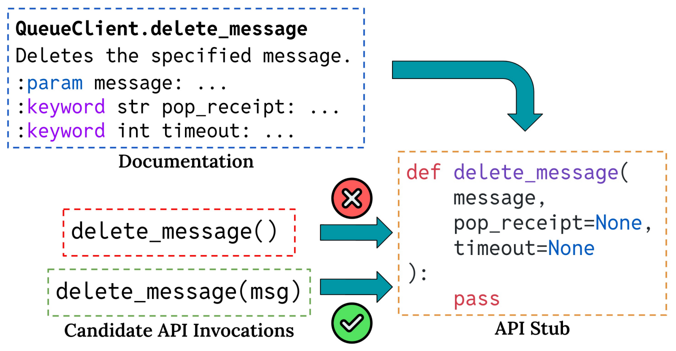
introduce `CloudAPIBench`, a benchmark designed to evaluate Code LLMs' abilities to invoke cloud APIs.

Composition. `CloudAPIBench` is a Python benchmark comprising 622 *synthetic tasks* to prevent data leakage with Code LLMs. Each task requires invoking a specific cloud API from providers like AWS and Azure, reflecting practical software engineering scenarios. Task prompts include imports, variable declarations, and a developer-style comment describing the API's purpose, stopping just before the API invocation. Figure 1 (left) illustrates a sample task and a model response (demonstrating an API hallucination). Figure 2 presents a detailed task distribution, showing that `CloudAPIBench` captures diverse API invocation scenarios to evaluate Code LLMs comprehensively.

API Frequency Annotations. `CloudAPIBench` also contains the *API frequency* for APIs, *i.e.*, how often they occur in The Stack v2 [14]. As The Stack v2 is one of the largest open code pre-training datasets, we assume that our API frequencies approximates the distribution of APIs in public sources. Hence, this can be used to explore the relationship between hallucination rates and API frequencies.

To enhance interpretability, we classify API frequencies into three categories: *Low* ($0-10$ occurrences), *Medium* ($11-100$), and *High* ($\geq$ 101). Since this treats APIs within the same class as identical, we minimize confounding factors (such as invocation complexity) by selecting diverse APIs within each class. This approach parallels the categorization of concepts based on popularity or pre-training frequencies [24], [25]. To our knowledge, this is the first granular analysis of a Code LLM's pre-training corpus.

Construction. We construct `CloudAPIBench` with the goal of scaling coverage to multiple APIs from various providers. First, we source API specifications from official documentation to index their correct usage. Next, we determine each API's frequency in The Stack v2 by counting function definitions and calls with the same names as the APIs in relevant files. We select APIs for `CloudAPIBench` while ensuring diversity of invocation complexity and frequency. Using Claude 3 Sonnet [26], we convert API descriptions into developer-style comments, and create prompts with necessary imports, declarations, and a descriptive comment before the API call.



```
QueueClient.delete_message
Deletes the specified message.
:param message: ...
:keyword str pop_receipt: ...
:keyword int timeout: ...
```

```
def delete_message(
    message,
    pop_receipt=None,
    timeout=None
):
    pass
```

```
delete_message()
```

```
delete_message(msg)
```

Fig. 3: **Valid API Invocation.** Using the API documentation, we create an API stub to capture correct usage. A candidate invocation is valid if it successfully binds to the stub. Here, `delete_message` requires *at least* one required argument for successful binding.

We provide elaborate details of this process in Appendix A.

Evaluation Metric. We introduce the **valid API invocation** metric, which verifies if an API is invoked according to its syntax. We obtain this syntax by tracking the API's arguments and whether they are required. The metric is computed as: we create a dummy function mirroring the API's signature (*i.e.*, API stub [27]). A candidate invocation is tested against this stub, and only successful bindings indicate validity. This evaluation method bypasses the intricacies of static analysis [11] and is more robust than string matching [20], ensuring reliable and scalable evaluations. Figure 3 illustrates this process.

C. *Evaluation & Results*

Models. We evaluate the following recent Code LLMs (and sizes) on `CloudAPIBench`: StarCoder2-{3B, 7B, 15B} [14], DeepSeekCoder-{1.3B, 6.7B, 33B} [30], Google CodeGemma-{2B, 7B} [31], IBM Granite-Code-{3B, 8B, 20B} [32] and GPT-4o (`gpt-4o-2024-05-13`) [29].

Inference. We use greedy decoding to generate code up to the first API call; this is evaluated for validity as detailed in Section II-B. This strategy is used throughout this work consistently. For chat models, we specify a system prompt indicating the model to generate only the API invocation.

Results. Table II presents the performance of all models on `CloudAPIBench` and HumanEval (for a reference of generic performance). Key observations include:

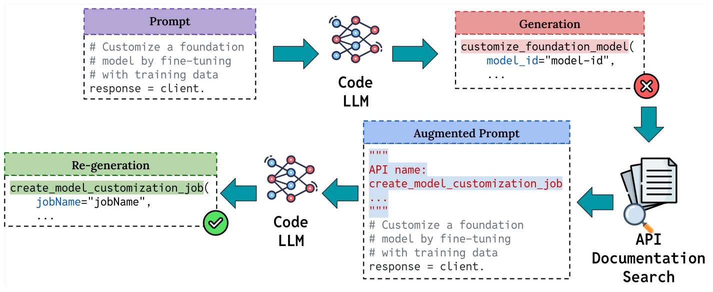
– *API Hallucinations.* All Code LLMs exhibit API hallucinations to a certain degree. These primarily occur due to (1) usage of non-existing APIs, (2) incorrect usage of the target API or, (3) usage of incorrect existing APIs. We explain these failure cases in Appendix B.

– *API Frequency Trends.* A strong correlation exists between API frequency and valid API invocations: high frequency APIs yield fewer hallucinations, while low frequency APIs result in more. While this is expected, this trend verifies the applicability of our API frequency annotations.

– *Low Frequency APIs.* Despite strong performance on high frequency APIs and generic benchmarks, all models

Model	HumanEval	CloudAPIBench		
	pass@1	*High Frequency*	*Medium Frequency*	*Low Frequency*
StarCoder2-3B	31.44	84.39	37.33	11.61
StarCoder2-7B	34.09	86.34	47.33	9.36
StarCoder2-15B	44.15	88.78	57.33	24.72
Google CodeGemma-2B	27.28	79.51	26.67	4.49
Google CodeGemma-7B	40.13	87.80	52.67	12.36
IBM Granite-Code-3B	−−	83.41	44.67	17.23
IBM Granite-Code-8B	−−	85.85	62.67	28.09
IBM Granite-Code-20B	−−	87.80	69.33	32.21
DeepSeekCoder-1.3B	32.13	79.02	22.67	5.24
DeepSeekCoder-6.7B	45.83	88.78	52.00	13.48
DeepSeekCoder-33B	52.45	90.24	70.00	34.83
GPT-4o	90.20	93.66	78.67	38.58

TABLE II: **Results on `CloudAPIBench`.** We present Valid API Invocation (%) results on `CloudAPIBench` for various Code LLMs, categorized by API frequency in The Stack v2. For comparison, we also include HumanEval [3] results from [28] and [29]. While Code LLMs excel on high-frequency APIs, their performance drops severely on low-frequency APIs, despite strong results on general programming tasks like HumanEval.



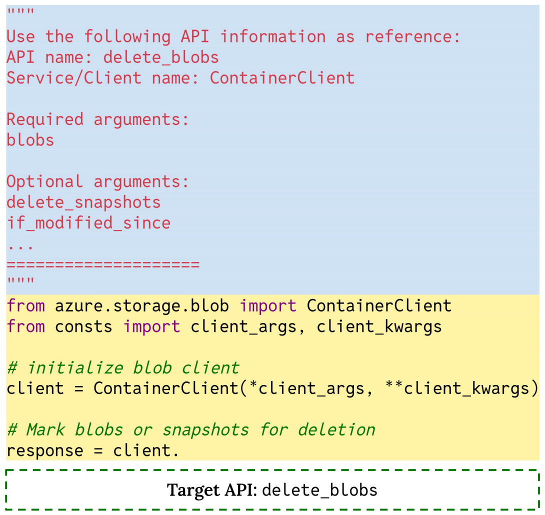
```
# Customize a foundation
# model by fine-tuning
# with training data
response = client.
```

```
customize_foundation_model(
    model_id="model-id",
    ...
```

```
"""
API name:
create_model_customization_job
...
"""
# Customize a foundation
# model by fine-tuning
# with training data
response = client.
```

```
create_model_customization_job(
    jobName="jobName",
    ...
```

Fig. 4: **DAG Overview.** Starting with a `CloudAPIBench` task, we sample an API invocation from the Code LLM. This is used to retrieve documentation for the matching APIs. We then augment the prompt with the documentation and re-trigger the model.



```
"""
Use the following API information as reference:
API name: delete_blobs
Service/Client name: ContainerClient

Required arguments:
blobs

Optional arguments:
delete_snapshots
if_modified_since
...
====================
"""
from azure.storage.blob import ContainerClient
from consts import client_args, client_kwargs

# initialize blob client
client = ContainerClient(*client_args, **client_kwargs)

# Mark blobs or snapshots for deletion
response = client.
```

Target API: `delete_blobs`

Fig. 5: **API Specification Augmentation.** Augmented prompt for the Oracle retriever with one retrieval. The “API Specification” (blue) contains the API name and a list of its required & optional arguments, providing an efficient summary of the documentation.

exhibit high hallucination rates for low frequency APIs. This disparity highlights the value of `CloudAPIBench` in pinpointing scenarios where Code LLMs are prone to hallucinate. Refer to Appendix C for a more detailed analysis.

III. Documentation Augmented Generation (DAG)

In this section, we see how DAG enhances performance on `CloudAPIBench`. We first outline the key components of DAG: augmentation design, retrieval index and retriever, in Section III-A. Subsequently, we discuss how different design choices affect downstream performance in Section III-B.

A. Setup

Overview. Following [15], [33], we implement an iterative pipeline for DAG. Starting with a prompt, the Code LLM generates a hypothetical API invocation. This invocation forms a query to retrieve documentation for similar APIs. The retrieved documentation is processed and appended to the original prompt, after which inference is re-triggered. This process is illustrated in Figure 4.

Query Formulation & Retrieval Index. Given a `CloudAPIBench` task, the Code LLM generates a candidate API invocation, which we process as the query. This query focuses solely on API-relevant keywords, excluding any distractor prompt content [15], [33], [34]. Our retrieval index includes all collected AWS and Azure APIs, identified using *keys* prepared similarly as the queries.

Retriever. We develop a retriever with configurable precision

Augmentation Design	Avg. Tokens	Valid API Inv. (%)
Base Model	−−	41.80
API Name Only	36.07	52.73
API Description	78.80	53.22
API Specification	52.57	86.82
API Desc. + API Spec.	94.55	87.14
Full Documentation	685.24	**88.75**

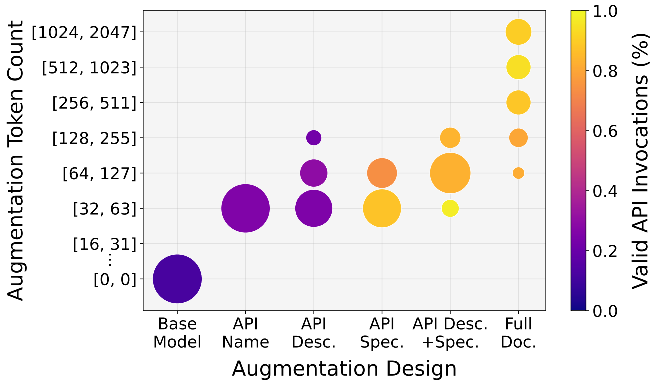


Fig. 6: **Augmentation Design Results. (Left)** Displays average tokens introduced per augmentation using the StarCoder2-3B tokenizer and average performance on `CloudAPIBench` for each augmentation design. **(Right)** Visualizes performance for *low frequency* APIs: the `y-axis` shows binned sequence lengths (exponential scale; capped at 2048), bubble color indicates performance, and bubble size indicates fraction of samples per bin. The improvements from API Specification are dramatic, though "Full Documentation" introduces too many tokens.

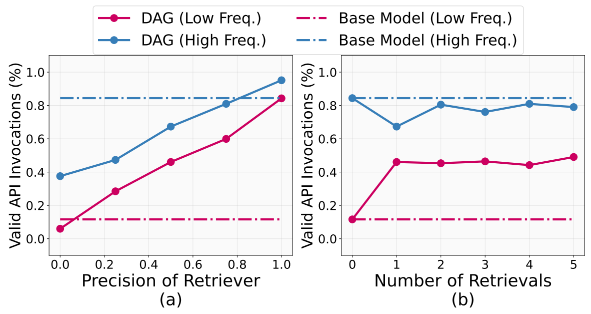


Fig. 7: **Precision and No. of Retrievals.** (a) While most low precision retrievers hurt performance on high frequency APIs, they may benefit low frequency APIs. (b) 1 retrieval hurts performance on high frequency APIs, but this is somewhat recovered as number of retrievals increases.

to study the effect of retrieval accuracy on `CloudAPIBench`. For an $x\%$ precision@k setting, we return k documents via BM25, ensuring that the target API's documentation is included $x\%$ of the time. This approach allows us to examine the impact of varying retrieval precision (x). We chose BM25 for its simplicity [11], [35], though our results are likely robust to different retrievers.

Augmentation Design. We prepend the retrieved documentation to the original prompt as a Python docstring after postprocessing. We test various augmentation strategies, each capturing different levels of API information and token count efficiencies: (1) API Name Only, (2) API Description, (3) API Specification, (4) API Description + Specification, and (5) Full Documentation. Figure 5 shows "API Specification" while additional details and examples are in Appendix D.

B. Experiments & Results

In this section, we perform ablations on various DAG components to analyze their impact on API hallucinations.

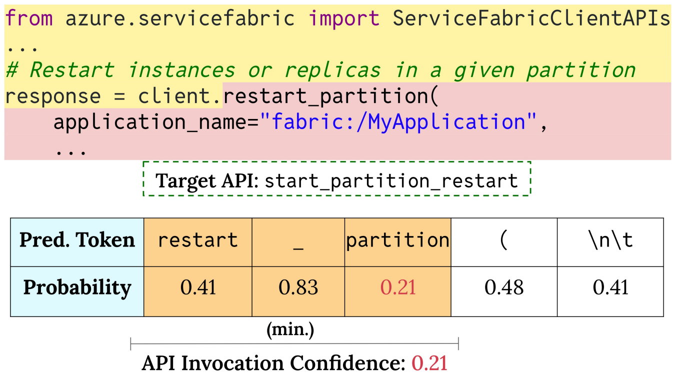
Experimental Setup. We present results from ablations on StarCoder2-3B. When testing a component (*e.g.*, retriever precision), other components (*e.g.*, number of retrievals) are held constant to isolate the effect. We also report the average valid API invocations across all tasks in `CloudAPIBench` for a concise performance measure, wherever indicated.

Method	*High Freq.*	*Low Freq.*	*Avg.*
Base Model	84.39	11.61	41.80
DAG	67.32	**46.07**	54.50
DAG + Index Lookup	**85.37**	35.96	**54.98**

TABLE III: **Index Lookup.** Triggering DAG only for non-existent APIs reduces unnecessary retrievals for high-frequency APIs, enhancing performance. However, this also induces slight regressions for low-frequency APIs. [*Avg.* indicates performance across all frequencies.]

Augmentation Design. Our objective is to determine the most useful and efficient information to retain from an API's documentation for augmentation. So, we use an Oracle retriever to fetch only one documentation; this guarantees that the relevant information is present *somewhere* in the documentation. Results are presented in Figure 6, showing valid API invocation rates and the number of tokens introduced per augmentation across all APIs. "API Name Only" and "API Description" do not significantly reduce hallucination rates, as they lack detailed API syntax. However, adding "API Specification" dramatically improves model performance ($41.80\% \rightarrow 86.82\%$ on average), indicating that detailed API specifications are crucial. While "Full Documentation" optimizes performance, it is highly token-inefficient (685.24 tokens per augmentation). "API Description + Specification" strikes an optimal balance between token efficiency and performance, so, we adopt this design for all subsequent experiments.

Precision of Retriever. Results are displayed in Figure 7a. Here, we retrieve one document and vary the retriever's precision. As anticipated, the API hallucination rate decreases as retriever



```
from azure.servicefabric import ServiceFabricClientAPIs
...
# Restart instances or replicas in a given partition
response = client.restart_partition(
    application_name="fabric:/MyApplication",
    ...
```

Target API: start_partition_restart

Pred. Token	restart	_	partition	(	\n\t
Probability	0.41	0.83	0.21	0.48	0.41

(min.)

API Invocation Confidence: 0.21

Fig. 8: **API Invocation Confidence.** We estimate the model's uncertainty by taking the minimum probability of the predicted API name tokens (orange in table).

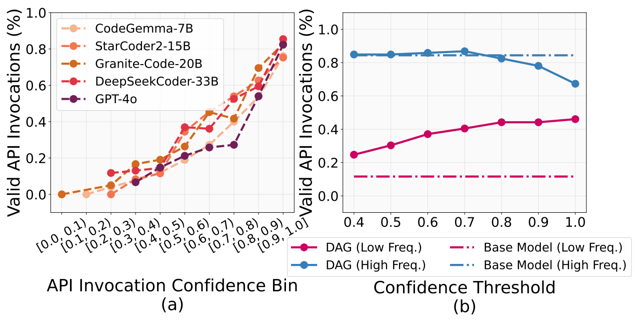


Fig. 9: **API Invocation Confidence Results.** (a) API invocation confidence scores are well-calibrated on CloudAPIBench for a range of Code LLMs. (b) By triggering DAG when the API invocation confidence is below a certain threshold, we can control regressions on high frequency APIs while maintaining good performance on low frequency APIs.

precision increases. Low frequency APIs show improvement over the base model even with low precision retrievers, while high frequency APIs require precision $> 80\%$ to match the base model's performance. Thus, at most precision levels, DAG induces higher hallucination rates for high frequency APIs compared to the base model (e.g., $84.39\% \rightarrow 67.32\%$ valid API invocations at 50% precision), underscoring Code LLMs' sensitivity to irrelevant augmentations [12].

Number of Retrievals. Here we maintain the retriever precision at 50%. Figure 7b illustrates our findings. For low-frequency APIs, one or more retrievals consistently enhance performance. Conversely, high-frequency APIs show a sharp decline with one retrieval, partially recovered with two or more. This indicates that irrelevant augmentations can lead to unexpected behavior in Code LLMs, *especially* when a single augmentation conflicts with the model's internal knowledge.

Discussion. Our experiments above show that DAG *significantly* reduces hallucinations for low frequency APIs. However, high frequency APIs may suffer performance drops with DAG due to irrelevant retrievals. This issue can potentially be resolved by allowing the Code LLM to use its internal knowledge for high frequency APIs, bypassing DAG; this forms the core of the next section.

IV. Improving DAG: When to Retrieve?

Given that suboptimal retrievers can increase hallucination rates with DAG, we investigate strategies to address this issue. By triggering DAG *selectively* – primarily when the Code LLM lacks knowledge of the target API – we can mitigate the negative impact of suboptimal retrievals, and allow the model to invoke APIs correctly using its internal knowledge. We discuss two strategies towards this here.

A. Index Lookup

Method. This simple technique verifies if the API name invoked during the first iteration generation exists in the API index. If not, the Code LLM is trying to call a non-existing API, and DAG provides the necessary references. Thus, DAG is not triggered for existing APIs; since this is likely to happen for high-frequency APIs, we expect fewer imprecise DAG triggers with this method.

Experimental Setup. As before, we perform ablations with StarCoder2-3B. We use a 50% precision retriever to retrieve one documentation.

Results. Table III presents the results. The index lookup method significantly reduces the regressions introduced by DAG for high-frequency APIs, even showing slight improvements over the base model. However, this gain comes at the expense of reduced retrievals for low-frequency APIs: sometimes, the model invokes an existing incorrect API or incorrectly invokes the target API, leading to more hallucinations compared to DAG. Overall, this method shows promise for enhancing DAG.

B. API Invocation Confidence

Background: LLM Calibration. Prior work has highlighted that LLM probabilities are well-calibrated, allowing for uncertainty estimation for various tasks [15], [36]–[38]. As such, leveraging a Code LLM's probabilities to predict potential hallucinations, we could selectively trigger DAG for those scenarios.

To quantify a Code LLM's uncertainty during API invocation, we define **API invocation confidence** as the *minimum* probability among all predicted API name (sub-)tokens (see Figure 8). This minimum captures minor uncertainties in API prediction better than other aggregators like the mean [15], [16]. The focus remains on the API name, not the entire invocation, as Code LLMs may show low confidence in tokens in the face of multiple alternatives (*e.g.*, constants in API arguments, *etc.*; this represents aleatoric uncertainty [39]).

Evidence from various Code LLMs, shown in Figure 9a, confirms their calibration for API invocation on CloudAPIBench. A strong positive correlation is observed between API invocation confidence and correct API usage, indicating that confidence levels can preemptively identify likely hallucinations (*i.e.*, they capture epistemic uncertainty [39]).

Method. We measure the API invocation confidence of the first iteration of generation, and if this is below a certain *fixed*

Model	Method	Retrieval Triggered (%)			Valid API Invocations (%)			
		High Freq.	*Med. Freq.*	*Low Freq.*	*High Freq.*	*Med. Freq.*	*Low Freq.*	*Avg.*
Google CodeGemma-7B	Base Model	0.00	0.00	0.00	87.80	52.67	12.36	46.95
	DAG	100.00	100.00	100.00	$61.95_{(-25.85)}$	$56.00_{(+3.33)}$	$46.44_{(+34.08)}$	$53.86_{(+6.91)}$
	DAG++	20.98	44.67	74.16	$88.29_{(+0.49)}$	$65.33_{(+12.67)}$	$43.07_{(+30.71)}$	$\mathbf{63.34}_{(+16.40)}$
StarCoder2-15B	Base Model	0.00	0.00	0.00	88.78	57.33	24.72	53.70
	DAG	100.00	100.00	100.00	$69.76_{(-19.02)}$	$58.67_{(+1.33)}$	$49.44_{(+24.72)}$	$58.36_{(+4.66)}$
	DAG++	20.98	43.33	70.41	$88.78_{(+0.00)}$	$58.67_{(+1.33)}$	$46.44_{(+21.72)}$	$\mathbf{63.34}_{(+9.65)}$
IBM Granite-Code-20B	Base Model	0.00	0.00	0.00	87.80	69.33	32.21	59.49
	DAG	100.00	100.00	100.00	$70.24_{(-17.56)}$	$63.33_{(-6.00)}$	$44.19_{(+11.99)}$	$57.40_{(-2.09)}$
	DAG++	15.12	29.33	66.29	$89.76_{(+1.95)}$	$71.33_{(+2.00)}$	$45.69_{(+13.48)}$	$\mathbf{66.40}_{(+6.91)}$
DeepSeekCoder-33B	Base Model	0.00	0.00	0.00	90.24	70.00	34.83	61.58
	DAG	100.00	100.00	100.00	$69.27_{(-20.98)}$	$64.00_{(-6.00)}$	$51.31_{(+16.48)}$	$60.29_{(-1.29)}$
	DAG++	20.49	30.67	59.55	$86.83_{(-3.41)}$	$71.33_{(+1.33)}$	$55.43_{(+20.60)}$	$\mathbf{69.61}_{(+8.04)}$
GPT-4o	Base Model	0.00	0.00	0.00	93.66	78.67	38.58	66.40
	DAG	100.00	100.00	100.00	$54.63_{(-39.02)}$	$53.33_{(-25.33)}$	$47.94_{(+9.36)}$	$51.45_{(-14.95)}$
	DAG++	3.41	9.33	50.56	$94.15_{(+0.49)}$	$82.00_{(+3.33)}$	$55.43_{(+16.85)}$	$\mathbf{74.60}_{(+8.20)}$

TABLE IV: **DAG++ Results.** We present the performance on `CloudAPIBench` and the (%) of retrieval triggers for high/low frequency APIs, with absolute improvements over the base model shown in subscript. It is noteworthy that DAG++ significantly reduces the frequency of retrievals for high frequency APIs while appropirately decides to retrieve for low frequency APIs; by smartly triggering retrieval, DAG++ attains top performance on `CloudAPIBench` for all models.

threshold, indicating the model's lack of knowledge about the target API, we trigger DAG to assist the model.

Experimental Setup. Towards finding an optimal configuration, we vary the threshold of API invocation confidence below which to trigger DAG, and measure the API hallucination rate for StarCoder2-3B. As before, we use a 50% precision retriever with one retrieved document.

Results. Figure 9b shows the relation between the confidence threshold and valid API invocations. As we raise the threshold, DAG is triggered more often, leading to a consistent reduction in hallucinations for low frequency APIs. Conversely, high frequency APIs remain largely unaffected until a certain point, beyond which irrelevant augmentations start causing hallucinations. The optimal threshold balances improved performance for low frequency APIs without significant regressions for high frequency APIs; for StarCoder2-3B, this optimal range is approximately $0.7 - 0.8$.

C. DAG++ & Discussion

Having seen the benefits of the above approaches, we now discuss how DAG can be effectively improved by combining these, *i.e.*, DAG++. In this method, we trigger DAG *iff* the API in the first iteration of generation does not exist in the index *OR* is being invoked with an API invocation confidence below a fixed threshold. We anticipate that this would help combine the benefits of both the discussed approaches.

Experimental Setup. We use a 50% precision retriever with one retrieval, consistent with previous experiments. Further, we fix the confidence threshold to be 0.8. Finally, to investigate the generalizability of our findings, we evaluate the largest models of all model families from Table II.

Results. The results are shown in Table IV. For each model, show how often retrieval is triggered and the resulting performance on `CloudAPIBench`. We make the following key observations:

– *Trigger of Retrievals.* We first examine how often retrieval is triggered with each method. The base model never triggers retrieval, DAG always does, and DAG++ selectively retrieves documentation. DAG++ exhibits a *strong negative correlation* between retrieval trigger frequency and API frequency: it triggers retrieval more often for low frequency APIs and less for high frequency APIs, aligning with the principle of retrieval *only when necessary*. For *e.g.*, with GPT-4o, DAG++ retrieves only 3.41% of the time for high frequency APIs indicating minimal need for documentation; conversely, retrieval is triggered 50.56% of the time for low frequency APIs, supplementing the model's limited knowledge with relevant documentation.

– *DAG v/s DAG++*. Table IV also shows the performances (and absolute improvements over the base model in subscript) of various models on `CloudAPIBench`. As noted in Section III, while DAG significantly boosts low frequency API performance, it degrades high frequency API performance. For instance, GPT-4o experiences a 39.02% drop in performance for high frequency APIs with DAG, highlighting the the model's sensitivity to irrelevant augmentations. DAG++ successfully mitigates this issue for high frequency APIs while maintaining or improving gains on low frequency APIs. Overall, DAG++ outperforms DAG indicating that selective retrieval of API documentation, that respects API frequency, aids performance on `CloudAPIBench`.

– *Generalizability.* All model families demonstrate similar enhancement trends with DAG++, despite architectural and training differences. This underscores the generalizability of the importance of selectively retrieving API documentation when Code LLMs lack API specific knowledge. Additionally, scaling trends with model sizes [40], [41] are evident: average performance monotonically improves with model size in Table IV. Finally, DAG++ reveals that larger models require fewer retrievals for optimal performance, suggesting they memorize API syntax (including low frequency) more efficiently.

V. Related Work

Program Synthesis & API Invocations. Code LLMs are actively being used for automatic program synthesis [30], [42]. Relevant to our study is API invocation generation [11], [43], often done on tool-usage benchmarks that do not account for the distribution of APIs in the public domain. We develop CloudAPIBench, a benchmark targeting cloud-based software engineering scenarios, that includes API frequency annotations, allowing for nuanced failure analyses and targeted improvements through DAG. Works such as [10], [11], [34], [44] also use documentation to improve API generation, but their evaluations do not capture the granularities discussed here.

LLM Hallucinations. LLMs may generate factually incorrect statements about concepts, diminishing their utility [45]–[47]. As such, several works have emerged to deal with this issue. Some works focus on hallucination detection by exploiting the well-calibrated nature of LLMs [36]–[38] and using model confidence scores [15], [16]. Closest to our work, [17], give a taxonomy of hallucinations for code generation. While they focus on identifying hallucinations with Code LLMs, we focus on mitigating API hallucinations using documentation.

Retrieval Augmented Generation (RAG). RAG supplements language models by retrieving from external data-stores [48]. Some studies use fixed algorithms for retrieval [11], [49], [50], while others adopt adaptive retrieval through special tokens [51] or model confidence scores [15]. In this work, we establish how to use selective retrieval effectively to mitigate API hallucinations with documentation.

VI. Conclusion & Future Work

In this work, we thoroughly investigate API hallucinations and demonstrate mitigation strategies for various Code LLMs. We introduce CloudAPIBench, a benchmark to measure API hallucinations for diverse AWS and Azure APIs, including API frequencies to categorize low, medium, and high frequency APIs. We adapt RAG with documentation (DAG) to inform Code LLMs about the correct syntax during inference. We discuss which parts of documentation are important and how various retrieval components affect hallucinations. While DAG significantly enhances low-frequency API performance, it can degrade high-frequency API performance with irrelevant retrievals. We tackle this issue by *selectively* triggering retrievals through index lookup and API invocation confidence thresholding, and combine these methods in DAG++ leading to top performance on CloudAPIBench across Code LLMs. Future research could extend CloudAPIBench for long-context evaluations, explore DAG beyond iterative generation, and improve DAG by enhancing Code LLMs' robustness to irrelevant augmentations.

Appendix

A. CloudAPIBench *Construction*

We follow a multi-step procedure to obtain synthetic evaluation tasks in Python for AWS and Azure APIs to include in CloudAPIBench. A summary of the process is shown in Figure 10. We describe each step in detail here.

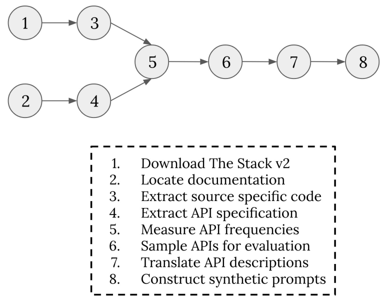

1
2
3
4
5
6
7
8
1. Download The Stack v2
2. Locate documentation
3. Extract source specific code
4. Extract API specification
5. Measure API frequencies
6. Sample APIs for evaluation
7. Translate API descriptions
8. Construct synthetic prompts

Fig. 10: **Summary of steps to construct CloudAPIBench.**

1) **Download The Stack v2.** We download The Stack v2 from its official repository on HuggingFace and SoftwareHeritage [14].
2) **Locate Documentation & Syntax.** We use boto3 1.34.108 for AWS and the Python package azure-sdk-for-python for Azure. For AWS, we use the mypy_boto3_builder tool [52] to create API stubs for all AWS APIs; this helps us obtain the list of APIs. We obtain the official documentation for each of these by scraping the boto3 webpage for the respective APIs. For Azure, the complete docstring in the source code for an API's defintion is its documentation.
3) **Extract source specific code.** We identify source specific code samples in The Stack v2 so that we restrict the count of API frequencies to only these. For AWS, source specific files are those that import one of {boto3, botocore} or contain one of {aws, boto, amazon} in the filepath. Similarly, Azure specific samples are those that import azure or contain azure in the filepath.
4) **Extract API specification.** For Azure, the complete documentation is available as a docstring in the respective function definitions for that API. Using tree-sitter, we parse the code files to obtain the list of APIs, their correct usages and complete docstrings for Azure, for as many APIs as possible. For AWS, we parse API stubs obtained using mypy_boto3_builder to curate the API specifications. This also serves as the index of APIs that we use in our experiments.
5) **Measure API frequencies.** Given the list of APIs for a source, we count the number of times functions with the name as an API are invoked or defined within the source specific code samples identified above. We use several heuristics to avoid edge cases and maintain reliability. Nevertheless, some noise may creep in and we acknowledge that this process is far from perfect. However, the findings based off of these API frequencies align with our expectations, indicating their reliability.
6) **Sample APIs for evaluation.** While sampling APIs for evaluation, we take care to ensure diversity with respect to API frequencies (uniform sampling from each frequency class as far as possible) and invocation complexity (within each frequency class, there should be uniform distribution of the number of required arguments required by APIs).

This ensures that CloudAPIBench is diverse and represents diverse software-engineering scenarios, from low to high frequency, and from APIs that are easy to invoke to those that require careful recall of API syntax. Further each API may appear in CloudAPIBench up to 3 times with different prompts.

7) **Translate API descriptions.** Each sample in CloudAPIBench contains a comment that expresses the intent to invoke the API. We obtain this comment by instructing Claude 3 Sonnet [26] to translate the documentation description of the API into 3 concise developer style comments. We use few-shot prompting to do this, and upon manual inspection of dozens of responses, find that Claude is able to do this task reliably. As such, we use Claude's responses as comments describing the intent to invoke the APIs, fixing any issues that were noticed manually.
8) **Construct synthetic prompts.** As the final step, for the selected APIs, we create synthetic prompts by creating an incomplete code sample: these start with relevant imports, contain necessary variable declarations, include the comment expressing the intent to invoke an API, and end just before the API invocation. Manual inspection revealed that in a few cases, multiple APIs may be suitable targets for a task, and in such cases we manually enumerate all the possible targets to the best of our knowledge.

B. Hallucination Categorization & Illustration

We classify API hallucinations into three broad categories:

1) **Usage of incorrect existing API**. This occurs when the Code LLM attempts to invoke an API that exists but does not fulfill the task.
2) **Invalid usage of target API**. This occurs when the Code LLM attempts to invoke the correct API but does so incorrectly due to an invalid configuration of arguments; here the model may either pass arguments that the API does not accept or not pass a correct combination of required and optional arguments.
3) **Usage of non-existing API**. This occurs when the Code LLM attempts to invoke an API that does not exist.

C. Analyzing Low Frequency API Failures

We look closer into the various modes of failure for low frequency APIs in Table V. We present this analysis for the largest model in each model family. As shown, most failures arise when the models try to invoke a non-existing API or use the target API incorrectly. This goes to show the lack of knowledge about low frequency APIs, and the propensity to hallucinate under these scenarios in Code LLMs.

D. Augmentation Designs

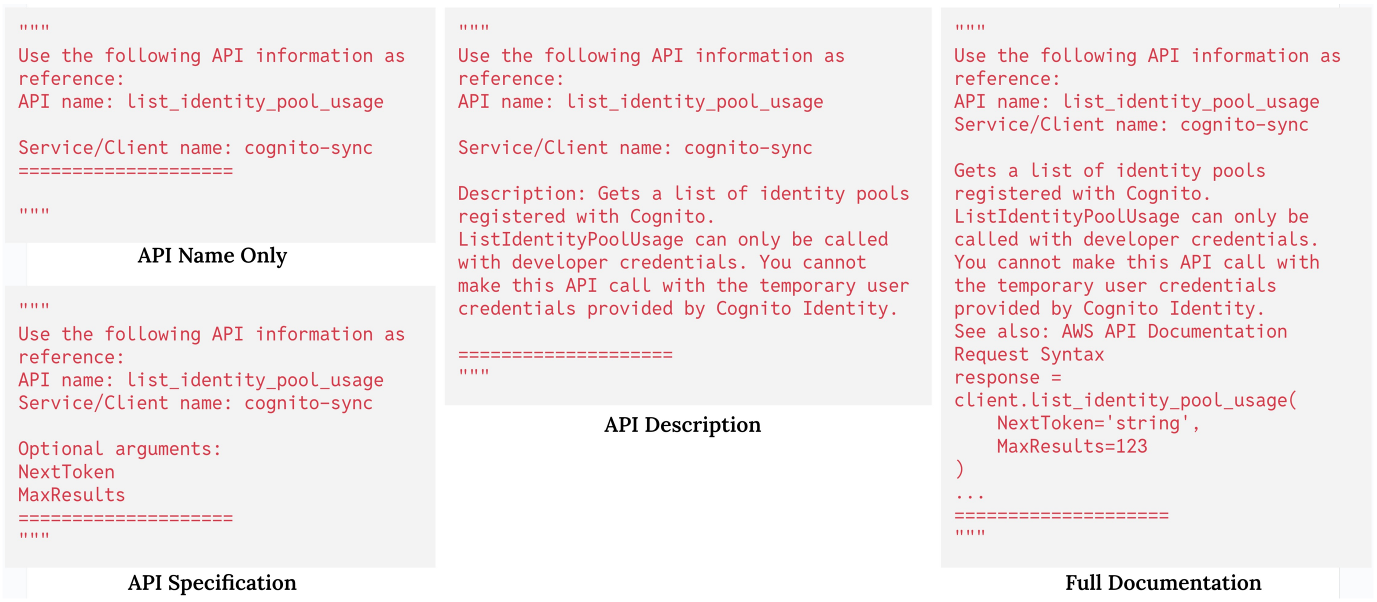
We define and illustrate various augmentation designs in this section.

- **API Name Only.** We include only the name of the retrieved APIs as augmentation. This can test if the Code LLM can invoke the correct API just by referencing its name during inference.
- **API Description.** We include the name and a short description of the API. For Azure the short description is the first sentence from the API's docstring, whereas for AWS the short description is the first 5 sentences from the API's documentation on the boto3 webpage. We choose 5 here as we found, in several cases, the first $2-3$ sentences to be irrelevant to the API's description.
- **API Specification.** This is a concise summary of the syntax of the API. It includes the name of the API and the list of required and optional arguments without specifying any descriptions of the arguments.
- **API Description + API Specification.** This includes the description as defined above along with the specification as discussed above.
- **Full Docstring.** This uses the entire collected documentation as augmentation. Since this can be arbitrarily large, especially for AWS documentation, we right-truncate the documentation up to 5000 characters before augmenting. This assumes that the necessary information to invoke the API is within the first 5000 characters.

We illustrate these strategies in Figure 11. We skip "API Description + API Specification" in the figure as it is a combination of "API Description" and "API Specification".

Model	Valid (%)	Invalid (%)	Invalid API Invocations Breakdown		
			Usage of incorrect existing	*Invalid usage of target*	*Usage of non-existing*
Google CodeGemma-7B	12.36	87.64	10.68	35.47	53.85
StarCoder2-15B	24.72	75.28	10.95	33.33	55.72
IBM Granite-Code-20B	32.21	67.79	17.13	23.76	59.12
DeepSeekCoder-33B	34.83	65.17	15.52	31.03	53.45
GPT-4o	38.58	61.42	13.41	33.54	53.05

TABLE V: **Results on `CloudAPIBench` for low frequency APIs.** We first show the fraction of valid and invalid API invocations for low frequency APIs for various models. The invalid API invocations are categorized into various types of failures. Notably, $> 50\%$ failures occur due to the models attempting to invoke non-existing APIs.



```
"""
Use the following API information as
reference:
API name: list_identity_pool_usage

Service/Client name: cognito-sync
====================

"""
```

API Name Only

```
"""
Use the following API information as
reference:
API name: list_identity_pool_usage
Service/Client name: cognito-sync

Optional arguments:
NextToken
MaxResults
====================
"""
```

API Specification

```
"""
Use the following API information as
reference:
API name: list_identity_pool_usage

Service/Client name: cognito-sync

Description: Gets a list of identity pools
registered with Cognito.
ListIdentityPoolUsage can only be called
with developer credentials. You cannot
make this API call with the temporary user
credentials provided by Cognito Identity.

====================
"""
```

API Description

```
"""
Use the following API information as
reference:
API name: list_identity_pool_usage
Service/Client name: cognito-sync

Gets a list of identity pools
registered with Cognito.
ListIdentityPoolUsage can only be
called with developer credentials.
You cannot make this API call with
the temporary user credentials
provided by Cognito Identity.
See also: AWS API Documentation
Request Syntax
response =
client.list_identity_pool_usage(
    NextToken='string',
    MaxResults=123
)
...
====================
"""
```

Full Documentation

Fig. 11: **API augmentation designs**. Illustrated for the AWS API: `list_identity_pool_usage`. "Full Documentation" is truncated to fit in the figure.

References

[1] S. Sivasubramanian, "Announcing new tools for building with generative ai on aws," Apr 2023, [Accessed: (June 15, 2024)]. [Online]. Available: https://aws.amazon.com/blogs/machine-learning/announcing-new-tools-for-building-with-generative-ai-on-aws/

[2] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," 2023.

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[4] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "Github copilot ai pair programmer: Asset or liability?" *J. Syst. Softw.*, vol. 203, no. C, Sep. 2023. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111734

[5] M. AI, "Introducing code llama, an ai tool for coding," Aug 2023, https://about.fb.com/news/2023/08/code-llama-ai-for-coding/.

[6] R. Salva, "Introducing ai-powered app dev with code customization from gemini code assist enterprise," Oct 2024, https://cloud.google.com/blog/products/application-development/introducing-gemini-code-assist-enterprise.

[7] D. Prakoso, "Announcing new tools for building with generative ai on aws," 2024, https://aws.amazon.com/blogs/aws/amazon-q-developer-now-generally-available-includes-new-capabilities_-to-reimagine-developer-experience/.

[8] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3586030

[9] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 70–79.

[10] S. Zhou, U. Alon, F. F. Xu, Z. Jiang, and G. Neubig, "Docprompting: Generating code by retrieving the docs," in *The Eleventh International Conference on Learning Representations*, 2023.

[11] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," 2023.

[12] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," in *Proceedings of the 40th International Conference on Machine Learning*, ser. ICML'23. JMLR.org, 2023.

[13] O. Yoran, T. Wolfson, O. Ram, and J. Berant, "Making retrieval-augmented language models robust to irrelevant context," in *The Twelfth International Conference on Learning Representations*, 2024.

[14] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024.

[15] Z. Jiang, F. Xu, L. Gao, Z. Sun, Q. Liu, J. Dwivedi-Yu, Y. Yang, J. Callan, and G. Neubig, "Active retrieval augmented generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 7969–7992. [Online]. Available: https://aclanthology.org/2023.emnlp-main.495

[16] N. Varshney, W. Yao, H. Zhang, J. Chen, and D. Yu, "A stitch in time saves nine: Detecting and mitigating hallucinations of llms by validating low-confidence generation," 2023.

[17] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, and L. Zhang, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[18] M. Zhang, O. Press, W. Merrill, A. Liu, and N. A. Smith, "How language model hallucinations can snowball," 2023.

[19] H. Ding, V. Kumar, Y. Tian, Z. Wang, R. Kwiatkowski, X. Li, M. K. Ramanathan, B. Ray, P. Bhatia, and S. Sengupta, "A static evaluation of code completion by large language models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, S. Sitaram, B. Beigman Klebanov, and J. D. Williams, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 347–360. [Online]. Available: https://aclanthology.org/2023.acl-industry.34

[20] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth *et al.*, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[21] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021.

[22] T. Liu, C. Xu, and J. McAuley, "Repobench: Benchmarking repository-level code auto-completion systems," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=pPjZIOuQuF

[23] K. Basu, I. Abdelaziz, S. Chaudhury, S. Dan, M. Crouse, A. Munawar, V. Austel, S. Kumaravel, V. Muthusamy, P. Kapanipathi, and L. Lastras, "API-BLEND: A comprehensive corpora for training and benchmarking API LLMs," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 12 859–12 870. [Online]. Available: https://aclanthology.org/2024.acl-long.694

[24] Y. Razeghi, R. L. Logan IV, M. Gardner, and S. Singh, "Impact of pretraining term frequencies on few-shot numerical reasoning," in *Findings of the Association for Computational Linguistics: EMNLP 2022*, Y. Goldberg, Z. Kozareva, and Y. Zhang, Eds. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, Dec. 2022, pp. 840–854. [Online]. Available: https://aclanthology.org/2022.findings-emnlp.59

[25] A. Mallen, A. Asai, V. Zhong, R. Das, D. Khashabi, and H. Hajishirzi, "When not to trust language models: Investigating effectiveness of parametric and non-parametric memories," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 9802–9822. [Online]. Available: https://aclanthology.org/2023.acl-long.546

[26] Anthropic, "Introducing the next generation of claude," 2024, https://www.anthropic.com/news/claude-3-family [Accessed: (March 4, 2024)].

[27] H. Zhu, L. Wei, V. Terragni, Y. Liu, S.-C. Cheung, J. Wu, Q. Sheng, B. Zhang, and L. Song, "Stubcoder: Automated generation and repair of stub code for mock objects," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–31, 2023.

[28] BigCode, "Big code models leaderboard - a hugging face space by bigcode," 2024, https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard [Accessed: (June 12, 2024)].

[29] OpenAI, "Hello gpt-4o," 2024, https://openai.com/index/hello-gpt-4o/ [Accessed: (June 19, 2024)].

[30] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024.

[31] C. Team, A. J. Hartman, A. Hu, C. A. Choquette-Choo, H. Zhao, J. Fine, J. Hui, J. Shen, J. Kelley, J. Howland, K. Bansal, L. Vilnis, M. Wirth, N. Nguyen, P. Michel, P. Choy, P. Joshi, R. Kumar, S. Hashmi, S. Agrawal, S. Zuo, T. Warkentin, and Z. e. a. Gong, "Codegemma: Open code models based on gemma," 2024. [Online]. Available: https://goo.gle/codegemma

[32] M. Mishra, M. Stallone, G. Zhang, Y. Shen, A. Prasad, A. M. Soria, M. Merler, P. Selvam, S. Surendran, S. Singh, M. Sethi, X.-H. Dang, P. Li, K.-L. Wu, S. Zawad, A. Coleman, M. White, M. Lewis, R. Pavuluri, Y. Koyfman, B. Lublinsky, M. de Bayser, I. Abdelaziz, K. Basu, M. Agarwal, Y. Zhou, C. Johnson, A. Goyal, H. Patel, Y. Shah, P. Zerfos, H. Ludwig, A. Munawar, M. Crouse, P. Kapanipathi, S. Salaria, B. Calio, S. Wen, S. Seelam, B. Belgodere, C. Fonseca, A. Singhee, N. Desai, D. D. Cox, R. Puri, and R. Panda, "Granite code models: A family of open foundation models for code intelligence," 2024.

[33] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "RepoCoder: Repository-level code completion through iterative retrieval and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 2471–2484. [Online]. Available: https://aclanthology.org/2023.emnlp-main.151

[34] A. Eghbali and M. Pradel, "De-hallucinator: Iterative grounding for llm-based code completion," 2024.

[35] X. Cheng, D. Luo, X. Chen, L. Liu, D. Zhao, and R. Yan, "Lift yourself up: Retrieval-augmented text generation with self-memory," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=lYNSvp51a7

[36] S. Kadavath, T. Conerly, A. Askell, T. Henighan, D. Drain, E. Perez, N. Schiefer, Z. Hatfield-Dodds, N. DasSarma, E. Tran-Johnson, S. Johnston, S. El-Showk, A. Jones, N. Elhage, T. Hume, A. Chen, Y. Bai, S. Bowman, S. Fort, D. Ganguli, D. Hernandez, J. Jacobson, J. Kernion, S. Kravec, L. Lovitt, K. Ndousse, C. Olsson, S. Ringer, D. Amodei, T. Brown, J. Clark, N. Joseph, B. Mann, S. McCandlish, C. Olah, and J. Kaplan, "Language models (mostly) know what they know," 2022.

[37] C. Si, Z. Gan, Z. Yang, S. Wang, J. Wang, J. L. Boyd-Graber, and L. Wang, "Prompting GPT-3 to be reliable," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=98p5x51L5af

[38] J. Li, T. Tang, W. X. Zhao, J. Wang, J.-Y. Nie, and J.-R. Wen, "The web can be your oyster for improving language models," in *Findings of the Association for Computational Linguistics: ACL 2023*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 728–746. [Online]. Available: https://aclanthology.org/2023.findings-acl.46

[39] Y. A. Yadkori, I. Kuzborskij, A. György, and C. Szepesvári, "To believe or not to believe your llm," 2024.

[40] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," 2020.

[41] B. Wang, W. Ping, P. Xu, L. McAfee, Z. Liu, M. Shoeybi, Y. Dong, O. Kuchaiev, B. Li, C. Xiao, A. Anandkumar, and B. Catanzaro, "Shall we pretrain autoregressive language models with retrieval? a comprehensive study," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 7763–7786. [Online]. Available: https://aclanthology.org/2023.emnlp-main.482

[42] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2023.

[43] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, L. Hong, R. Tian, R. Xie, J. Zhou, M. Gerstein, dahai li, Z. Liu, and M. Sun, "ToolLLM: Facilitating large language models to master 16000+ real-world APIs," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=dHng2O0Jjr

[44] D. Zan, B. Chen, Y. Gong, J. Cao, F. Zhang, B. Wu, B. Guan, Y. Yin, and Y. Wang, "Private-library-oriented code generation with large language models," 2023.

[45] A. Mishra, A. Asai, V. Balachandran, Y. Wang, G. Neubig, Y. Tsvetkov, and H. Hajishirzi, "Fine-grained hallucination detection and editing for language models," 2024.

[46] H. Kang, J. Ni, and H. Yao, "Ever: Mitigating hallucination in large language models through real-time verification and rectification," 2023.

[47] H. Lee, S. Joo, C. Kim, J. Jang, D. Kim, K.-W. On, and M. Seo, "How well do large language models truly ground?" 2023.

[48] A. Asai, S. Min, Z. Zhong, and D. Chen, "Retrieval-based language models and applications," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 6: Tutorial Abstracts)*, Y.-N. V. Chen, M. Margot, and S. Reddy, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 41–46. [Online]. Available: https://aclanthology.org/2023.acl-tutorials.6

[49] Y. Wang, P. Li, M. Sun, and Y. Liu, "Self-knowledge guided retrieval augmentation for large language models," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 10 303–10 315. [Online]. Available: https://aclanthology.org/2023.findings-emnlp.691

[50] W. Shi, S. Min, M. Yasunaga, M. Seo, R. James, M. Lewis, L. Zettlemoyer, and W.-t. Yih, "REPLUG: Retrieval-augmented black-box language models," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, K. Duh, H. Gomez, and S. Bethard, Eds. Mexico City, Mexico: Association for Computational Linguistics, Jun. 2024, pp. 8371–8384. [Online]. Available: https://aclanthology.org/2024.naacl-long.463

[51] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi, "Self-RAG: Self-reflective retrieval augmented generation," in *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*, 2023. [Online]. Available: https://openreview.net/forum?id=jbNjgmE0OP

[52] YouType, "mypy_boto3_builder," https://github.com/youtype/mypy_boto3_builder/, 2024.