Synthetic APIs: Enabling Language Models to Act as Interlocutors Between Natural Language and Code

Ryan Mullins
Google Research
ryanmullins@google.com

Michael Terry
Google Research
michaelterry@google.com

Abstract

Large language models (LLMs) can synthesize code from natural language descriptions or by completing code in-context. In this paper, we consider the ability of LLMs to synthesize code, at inference time, for a novel API not in its training data, and specifically examine the impact of different API designs on this ability. We find that: 1) code examples in model training data seem to facilitate API use at inference time; 2) hallucination is the most common failure mode; and 3) the designs of both the novel API and the prompt affect performance. In light of these findings, we introduce the concept of a ***Synthetic API****: an API designed to be used by LLMs instead of by humans. Synthetic APIs for LLMs offer the potential to further accelerate development of natural language interfaces to arbitrary tools and services.*

Keywords: Large language models, Code synthesis, API design, Human-AI interaction

1. Introduction

Recent work has shown the ability of large language models (LLMs) to synthesize code from natural language descriptions and existing code (Brown et al., 2020; Chen et al., 2021; “GitHub Copilot”, 2021; Li et al., 2022). These capabilities have been demonstrated with models trained on a general corpus of web content, which may include source code (Austin et al., 2021; Brown et al., 2020), as well as models explicitly trained for code synthesis (Chen et al., 2021; Li et al., 2022).

One specific use case for code synthesis is to translate high-level, natural language requests into code that makes use of a specific application programming interface (API) or library. For example, a developer may wish to write code to visualize data using a library they have never used. In this context, a developer may be able to clearly articulate the desired outcome, but otherwise be unable to write the desired code without external resources (e.g., documentation, Stack Overflow examples, peer assistance). In these circumstances, LLM-powered code synthesis can reduce the need to turn to external resources, while generating code that makes use of the surrounding context.

In this research, we are interested in the ability of LLMs to transform natural language requests into code for novel APIs, which we define as APIs not within the training data. Instead, the model is exposed to the API at inference time via prompt engineering, as opposed to other means, such as pre-training or fine-tuning.

The ability to use associations between natural language requests and arbitrary functions in a novel API is an important use case: one cannot expect all APIs to be represented in a model’s training data (e.g., new APIs are continually being created). The ability to use a new API is also of inherent interest to the study of LLMs: measuring how well, and how quickly, an LLM can make use of an API is a useful way to compare different models’ capabilities. Finally, it may be the case that some API designs are easier for LLMs to use than others. If this is the case, one can design APIs expressly for use by an LLM (as opposed to for use by humans).

In this paper, we report results from two experiments that examine the models’ abilities to synthesize code using novel APIs. We further consider whether some API designs are easier for an LLM to use than others. We call this latter concept a *Synthetic API (SynAPI)* to convey the idea of an API intentionally designed for use by an LLM.

In our experiments, we vary two dimensions: 1) API naming conventions (verbose, abbreviated, random) and 2) the number of examples provided for each API function (1, 3, or 7). We examine performance across these dimensions for four models, where these models vary in size (137B, 175B, and 540B parameters) and training data (more specifically, one model, PaLM-Coder, includes significant code examples in its training data, while the other three contain significantly fewer examples). The experiments examine code synthesis performance for APIs we designed to act as programmatic interfaces to the `git` version control system.

We summarize this paper's contributions as:

- We introduce and test the concept of a SynAPI, an API designed for use by an LLM, rather than a human.
- We report results from two experiments testing four different LLMs' abilities to use four different API designs. Our results suggest that:
  - Models explicitly trained for code synthesis are likely to outperform other models for this task, suggesting the value of training data for using new APIs.
  - The most frequent error mode for the models is synthesizing code that includes functions not defined in the API (i.e., hallucination, which we refer to as "out-of-API" errors for this task).
  - API design choices *do* impact model performance. Surprisingly, for two models tested, an API design employing random function names (i.e., names with no meaning in the target domain) outperformed domain-specific function names.
  - Within a prompt, increasing the number of example uses of a function can increase the likelihood of correct usage, but to a limit. Too many examples seem to bias the model to particular types of output.

These results demonstrate that LLMs can make use of novel APIs at inference time, and that API design choices can affect their performance. These results additionally suggest the value in further researching SynAPIs, to understand how APIs can be optimally designed for use by an LLM.

2. Related Work

Large language models (LLMs) have demonstrated the ability to generate useful content in a wide variety of contexts (Brown et al., 2020; Chowdhery et al., 2022; Thoppilan et al., 2022), including code synthesis (Austin et al., 2021; E. Jiang et al., 2022). At the most basic level, these models generate highly probable text-based content given an input string, with the likelihood of the generated text dependent upon the training data of the model. For example, given the input "The opposite of hot is", an LLM is likely to produce the word "cold" (plus additional content).

One of the defining features of recent LLMs is the ability to customize the model using *prompt programming*, or the careful crafting of the text input to produce a particular outcome (Brown et al., 2020). For example, to prime the model to translate natural language requests to HTML, it may be enough to format the input as a series of examples (referred to as *few-shot prompting*). For instance, in this (trivial) example, the input is designed to produce the HTML for a cancel button:

```
Request: An OK button.
HTML: <button>OK</button>.

Request: A Cancel button.
HTML:
```

From this, the model is likely to produce the HTML string "`<button>Cancel</button>`".

Prompt programming lowers the barrier to customizing an LLM for specific tasks, such as code synthesis (Austin et al., 2021; Brown et al., 2020; E. Jiang et al., 2021). In this paper, we're particularly interested in the opportunity prompt programming and LLMs provide for creating natural language interfaces to APIs that do not exist in the model's training data. This capability could allow software developers to quickly develop custom natural language interfaces to arbitrary APIs and services without incurring additional training and operations costs.

In the realm of code synthesis, a number of efforts have explored the ability for LLMs to produce code using 1) models not intentionally trained for this purpose (Austin et al., 2021), 2) models later fine-tuned for code synthesis (Chowdhery et al., 2022), and 3) models purposely trained for code synthesis (Chen et al., 2021; "GitHub Copilot", 2021; Li et al., 2022). These efforts clearly demonstrate the ability for LLMs to synthesize code from natural language descriptions (as well as from existing code). Our specific focus is on an LLM's ability to translate natural language requests to code for an API that does not exist in its training data. We consider models not specifically trained for code synthesis (GPT-3 [Brown et al., 2020], LaMDA [Thoppilan et al., 2022], PaLM [Chowdhery et al., 2022]), as well as a model specifically trained for code synthesis (a version of PaLM [Chowdhery et al., 2022]).

When an LLM synthesizes code for an API that does not exist in its training data, it is effectively deriving new associations at inference time. Recent research has demonstrated how LLMs can be augmented with non-differentiable memory to increase their ability to learn at inference time (Wu et al., 2022). This prior work shows great promise in supporting our particular use case. However, in this paper, we consider the ability for LLMs to use novel APIs without additional fine-tuning or memory augmentation.

Code synthesis can fail in many ways, making assessment of code synthesis capabilities challenging (Allamanis et al., 2018). For example, synthesized code may fail to execute because of a small syntax error that a human (or other automated tool) could easily correct. In this paper, we are interested in the ability for a model to synthesize code for a new API, and thus are interested in how well the model can derive associations to new function names. We are also interested in the *types* of errors it makes in these conditions. For example, when the model fails to use the correct function name in the novel API, does it use another function name from the API, or does it hallucinate a new one?

3. Testing API Usability

To test API usability, we introduce the concept of a Synthetic API (SynAPI)—an API explicitly designed for use by an LLM. These APIs can be thought of as domain-specific languages that encapsulate complex API surfaces into a form that can be used at inference time by an LLM. For the purposes of this work, we focus on SynAPIs that are written for Python 3.

3.1. API Design Choices

In this research, we focus on APIs designs that vary along two dimensions:

- **Faithfulness**: An API's faithfulness is the extent to which the API's function and argument names honor the names used in the target domain (where the target domain could be an existing tool, service, and/or other API).
  - A *faithful* API will use the same names that are used in the target domain (e.g., for an API to interface with the `git` program, it may use the function name "clone" for the clone operation).
  - A *semi-faithful* API will use semantically similar names to concepts in the target domain, but will vary from the canonical terminology (e.g., for the `git` clone operation, a semi-faithful API may use the function names "copy" or "download").
  - An *unfaithful* API will not use any of the names from the target domain (e.g., it may use random function names, like "shoe" to represent `git`'s clone operation).
- **Brevity**: An API can make use of identifiers of varying length, which we define as *brevity*. For example, a function name may consist of one or two letters, or a long, descriptive name reflective of the terminology used in the target domain.

Faithfulness and brevity have been studied in both software engineering and LLM contexts and have implications for maintenance costs and usability (Ahn et al., 2022; Attanasio et al., 2022; Flauzino et al., 2018; Kaur and Fuad, 2010). There are many other design dimensions that could be evaluated in future research.

3.2. Measuring Performance

In our experiments, the primary measure of performance we consider is *success rate*: the proportion of synthesized code that correctly executes compared to the ground truth.

3.3. Categorizing Failure Modes

When models fail to produce the desired output, it is useful to understand how they fail. In this research, we consider three potential outcomes for the synthesized code: 1) a correct API call, 2) an API call using an incorrect function or argument name, where that name is defined in the target API (i.e., an "in-API" error), or 3) an API call using a function or argument name not defined in the API (i.e., an "out-of-API" error, often called "hallucinations").

Here, we focus on APIs targeting Python 3 with named argument function calls. This calling convention offers more specific failure modes. Out-of-API function names will cause a `NameError` when the Python interpreter tries to call a function that is not defined in the environment. Out-of-API argument names will cause a `TypeError` in the Python interpreter. In-API argument name errors will likely cause a `TypeError` when the interpreter calls a function with a named argument not defined in its signature, but may also succeed if functions use the same argument names.

The model may also generate incorrect argument values by extracting incorrect data from the input, or by hallucinating values. Incorrect values are always failures, but may provide more detailed error information, such as a `SyntaxError` if incorrectly formatted, a `TypeError` if the wrong type, or a `ValueError` if it exceeds an allowed range.

4. Experiment 1: Using Git

4.1. Task

Git is the world's most commonly used version control system. It is also one of the more difficult API surfaces to understand (Perez De Rosso and Jackson, 2013). In this experiment, we explore the ability of an LLM to successfully call an API designed to interface with the `git` program. We have chosen to focus on a subset of `git` commands, categorized below according to the "Git Reference Manual" (2022), to make the task more tractable in an experimental setting.

- Getting and creating a project: `git clone`, `git init`.
- Snapshotting: `git add`, `git commit`, `git mv`, `git reset`, `git rm`, `git status`.
- Branching and merging: `git checkout`, `git merge`, `git tag`.
- Sharing and updating projects: `git fetch`, `git pull`, `git push`.

This subset of `git` commands varies in the number of arguments they take (0, 1, or 2) and whether or not those arguments are required.

4.2. API Designs

We implemented four APIs in Python 3 (Table 1). All designs have the same structure with one function for each of the target `git` commands, and arguments for each of the most common command parameters. The four API designs differ only in their function and argument names.

- The Faithful API uses the exact same function and argument names as `git`.
- The Terse API uses one- or two-character abbreviations of the `git` names, so `clone` becomes `cl`, etc. We did not change the `rm` or `mv` commands as they already meet this design.
- The Generic API uses terms that are not specific to any one version control system, but are still evocative of the functionality provided. For example, instead of `clone`, this API uses `download_project_from_server`.
- The Random API was designed to use words that have no connection to any version control system. It was created using an English-language random word generator to get a replacement for each unique function and argument name in the Faithful API. For example, the function name `fetch` became `crackpot`.

4.3. Models

We compared performance between four models that use a decoder-only transformer arhcitecture:

Table 1. SynAPI designs used in this experiment.

API Design	Faithfulness	Brevity
Faithful	Faithful	Average
Terse	Unfaithful	High
Generic	Semi-Faithful	Low
Random	Unfaithful	Low

- **PaLM**, a 540B parameter LLM from Google (Chowdhery et al., 2022), trained on a mix of web content and source code.
- **PaLM-Coder** is a PaLM-540B variant explicitly trained for code synthesis using a Python source code data set (Chowdhery et al., 2022).
- **LaMDA**, a 137B parameter LLM from Google (Thoppilan et al., 2022), trained to specialize in safe, factually grounded dialog via a corpus of public dialog data and web content.
- **GPT-3** (`text-davinci-002` version), a 175B parameter LLM from OpenAI with training data explicitly constructed to exclude source code (Brown et al., 2020).

LaMDA was specifically included to compare the performance of general-purpose text completion models (GPT-3 and PaLM) against one trained for a specific type of natural language text completion (in this case, dialog). PaLM-Coder was chosen to compare the performance of a code synthesis model against general-purpose text completion models.

4.4. Prompt Designs

We used prompt engineering to prime the models to synthesize code for the novel APIs. Prompt engineering provides a fast on-ramp for experimentation, supports the target use cases (i.e., synthesizing code for a novel API), and was the only method supported by all models.

To test the API designs, we developed a baseline prompt design consisting of manually curated examples (see the next section for an excerpt of the prompt design). Each example consists of a natural language input and API call pair. The natural language input is formatted as a Python comment immediately preceding the API call. The prompt includes one example for each function call parameterization, totalling 21 examples. As an example, the `git mv` command requires a source and destination parameter, so there is a single example of this API call present in the prompt. In contrast, the `git reset` command can operate in three different modes, so there are three examples of this API call in the prompt, one for each mode.

Python has flexibility in its argument passing style—positional arguments, named arguments, or both. In this work, all of the code in the baseline prompt uses only the named arguments convention.

From this baseline prompt design, we derived four different prompts, one for each API design.

4.5. Test Inputs

We manually created a set of 105 natural language inputs, which represent the user's objective, to use as test inputs. Examples of these test inputs are: "create a new project", "pull down the latest changes", and "add the .py files". Each test input is concatenated to the end of the prompt as a Python comment, shown in an abbreviated example from the Faithful API below. These prompt-plus-test-input formulations are fed to the LLM. Each test input is associated with a ground truth `git` command. Note that none of the test inputs are contained in the baseline or derived prompt designs.

```
# init
init()

# (Other function examples...)

# pull changes
pull()

# pull changes from dev
pull(branch="dev")

# (Other function examples...)

# add my_file.py
add(pathspec="my_file.py")

# (Other function examples...)

# checkout feature_branch
checkout(branch="feature_branch")

# checkout feature_branch and create
# it if it does not exist
checkout(branch="feature_branch",
         create=True)

# (Other function examples...)

# tag with my_tag
tag(name="my_tag")

# create a new project
```

4.6. Experimental Procedure

The prompt-plus-test-input formulation for each of the 105 test inputs, in each of the API designs, was fed to each of the four models. All models were run with `temperature = 0` to avoid non-deterministic behavior. The text output generated, referred to below as a "code generation," was collected for analysis.

We implemented each function in the four API designs in Python, so that calling the function in the Python interpreter generates a correctly parameterized `git` command string.

We test performance by 1) executing the code generation in a Python 3 environment (loaded with the target API), and 2) comparing the output to the ground truth `git` command for the corresponding test input.

For each code generation, we collect measures of correctness and failure mode. A code generation may contain multiple errors that each would cause the execution to fail in the Python interpreter. For example, a code generation may contain an incorrect but in-API function name, and an out-of-API argument name, but pass a valid value to that function. For each code generation, we record the most severe error type for each error category (function name, argument name, argument value). We order severity from least to most severe as: correct, in-API, and out-of-API.

4.7. Results

Table 2 summarizes the success rate of the different API designs across models. The PaLM-Coder model outperforms all other models, suggesting the importance of training data on the ability for an LLM to synthesize code using a novel APIs introduced at inference time. Surprisingly, the two PaLM model variants have the highest success rates on the Random API design, whereas this API design is the worst or tied-for-worst design for the other two models.

Per-`git` command success rates are summarized in Table 3. We observe that LaMDA performs slightly differently than the other models. Looking a bit closer, we observe that LaMDA performs best with function calls requiring no arguments, no required arguments, or a fewer number of arguments than the function defines (i.e., in instances when default argument values can be used). Trends are unclear when looking at this data across API designs, independent of model.

Table 4 lists the prevalence of the two failure modes examined (in-API and out-of-API). Out-of-API errors were by far the dominant failure mode, accounting for 48.82% of failures (across all models and API designs), compared to 17.32% for in-API errors. PaLM-Coder

Table 2. Percent correct output by API design and model.

API Design	PaLM	PaLM-Coder	LaMDA	GPT-3	*Mean*
Faithful	39.05	**44.76**	30.97	40.95	38.93
Terse	39.05	**40.95**	15.74	36.19	32.98
Generic	33.33	**56.19**	31.48	51.43	**43.11**
Random	48.57	**59.05**	4.46	36.19	37.07
Mean	40.00	**50.24**	20.66	41.19	38.02

Table 3. Percent correct by Git command for models and API designs.

			Models				API Designs			
Command	# of Args.	# of Req. Args.	PaLM	PaLM-Coder	LaMDA	GPT-3	Faithful	Terse	Generic	Random
`init`	0	0	**25.00**	**25.00**	20.00	15.00	30.00	0.00	**40.00**	15.00
`clone`	1	1	50.00	**66.67**	8.00	45.83	**58.33**	28.00	29.17	54.17
`fetch`	1	0	29.17	41.67	42.31	**45.83**	32.00	20.83	**83.33**	24.00
`pull`	1	1	10.42	14.58	5.77	**22.92**	12.00	10.42	**26.53**	4.08
`push`	1	0	**40.62**	**40.62**	28.12	37.50	40.62	31.25	**50.00**	25.00
`add`	1	1	50.00	**67.50**	21.74	60.00	**68.29**	41.46	43.90	41.86
`commit`	0	0	31.25	**50.00**	47.06	**50.00**	25.00	43.75	**75.00**	35.29
`mv`	2	2	43.75	**56.25**	12.50	43.75	31.25	**56.25**	12.50	56.25
`reset`	1	0	15.00	17.50	7.14	**25.00**	**26.83**	21.95	15.00	0.00
`rm`	1	1	62.50	**81.25**	17.65	50.00	**64.71**	56.25	25.00	62.50
`status`	0	0	35.00	**55.00**	20.00	30.00	25.00	40.00	25.00	**50.00**
`checkout`	2	1	63.33	**75.00**	30.65	45.00	45.90	48.33	**60.00**	59.02
`merge`	1	1	8.33	**25.00**	4.00	20.83	12.50	12.50	16.00	**16.67**
`tag`	1	1	77.50	**85.00**	29.27	75.00	58.54	52.50	**77.50**	**77.50**

consistently has the lowest out-of-API error rate. Trends across models for in-API error rates are less clear.

Examining these data across API designs, the Terse API design was the most likely to induce out-of-API errors across models, with a mean rate of 54.49%. The Generic API design was the least likely to induce out-of-API errors, with a mean rate of 45.53%, followed closely by the Random design at 46.31%. These mean rates suggest that LLMs may derive utility from more verbose API designs.

Table 5 lists the distribution of errors organized by `NameError` and `TypeError` (recall that `NameError` errors denote an attempt to call a function not defined in the environment, while `TypeError` errors can denote incorrect argument names). PaLM-Coder has the lowest `NameError` rates among all models, with trends for `TypeError` rates less clear across models. Looking across API designs, `TypeError` rates are consistent across models with the Random being least likely, followed by Terse, Generic, and Faithful. While possible, no generation contained a `SyntaxError`.

We examine the differences in `NameError` rates by considering the information density of the function names in each API design. We use token count as a proxy for information density. The Faithful, Terse, and Random designs all have very low mean token counts (1.14, 1, 1.14 tokens per function name respectively), whereas the Generic design has 4.33 tokens per function name. Mean token count aligns with the mean success rate for each API design (see Table 2).

5. Experiment 2: Few-shot prompting

Prior research has shown that LLMs perform differently when provided with more examples in a prompt engineering context (Logan IV et al., 2021). Our first experiment (above) shows that certain `git` commands were difficult for all models to use across all API designs using single-shot prompting (see Table 3). Given these data, we conducted a second experiment that used multiple examples per API function, using a subset of the worst-performing `git` commands from the first experiment. We used the same API designs and experimental procedure in this experiment as in the first.

5.1. Selecting the command subset

We chose to focus on the three `git` commands that had the lowest mean success rates across models in our first experiment: `fetch`, `merge`, and `reset`.

Table 4. In-API and out-of-API (i.e., hallucination) rates, inclusive of function and argument names, across models by API design.

	In-API Error Rate					**Out-of-API (Hallucination) Error Rate**				
API Design	PaLM	PaLM-Coder	LaMDA	GPT-3	*Mean*	PaLM	PaLM-Coder	LaMDA	GPT-3	*Mean*
Faithful	29.52	23.81	17.70	**12.38**	20.85	44.76	**41.90**	53.98	55.24	48.97
Terse	17.14	16.19	20.37	**13.33**	16.76	45.71	**40.95**	71.30	60.00	54.49
Generic	20.95	**10.48**	11.11	11.43	**13.49**	57.14	**29.52**	58.33	37.14	**45.53**
Random	28.57	23.81	**8.04**	12.38	18.20	21.90	**19.05**	92.86	51.43	46.31
Mean	24.05	18.57	14.30	**12.38**	17.32	42.38	**32.86**	69.12	50.95	48.82

Table 5. `NameError` and `TypeError` rates across models by API design.

	`NameError` Rate					**`TypeError` Rate**				
API Design	PaLM	PaLM-Coder	LaMDA	GPT-3	*Mean*	PaLM	PaLM-Coder	LaMDA	GPT-3	*Mean*
Faithful	39.05	**36.19**	53.98	46.67	43.97	19.05	16.19	**4.42**	13.33	13.45
Terse	44.76	**37.14**	69.44	58.10	52.36	3.81	4.76	2.78	**1.90**	3.31
Generic	53.33	**26.67**	56.48	33.33	**42.45**	9.52	6.67	**5.56**	5.71	6.86
Random	20.95	**19.05**	91.96	51.43	45.85	0.95	**0.00**	1.79	**0.00**	**0.68**
Mean	39.52	**29.76**	67.96	47.38	46.16	8.33	6.90	**3.64**	5.24	6.08

5.2. Models

Given the overall performance observed in the first experiment, this second experiment was only run on the PaLM model.

5.3. Prompt Design

In this experiment, we want to compare the performance difference with our first experiment, the results from which we use as a reference. We revised our prompt designs to create 1-shot, 3-shot, and 7-shot examples for each API.

PaLM imposes a limit on the number of tokens (500) that can be included in prompts to bound inference performance (Chowdhery et al., 2022). This can be challenging when using prompt engineering techniques, as we chose to here, since all examples plus the novel input must fit within the context window.

Due to this token limit, we had to vary the prompt construction based on shot-size classes to fit in the context window. For 1-shot and 3-shot prompting, we were able to use the exact same prompt design as in our first experiment, where every example for every API function was included in one prompt for that shot-size class. For 7-shot prompting, we were forced to separate the examples for `fetch`, `merge`, and `reset` into different prompts because they would overflow the 500-token context window. This means that prompts for each these commands *only* contain examples for that specific command (and no other command). However, each command has three parameterizations (i.e., different API call permutations), so each 7-shot prompt included 21 examples.

Table 6. Success rate by Git command by the number of examples (shots) included in the prompt for each API

Command	Reference	1-Shot	3-Shot	7-shot
`fetch`	29.17	20.83	**33.33**	16.67
`merge`	8.33	45.83	**79.17**	31.94
`reset`	15.00	27.50	**45.00**	13.33
Mean	17.50	31.39	**52.50**	20.65
Δ	-	+13.89	**+35.00**	+3.15

5.4. Test Inputs

We filtered the original 105 test inputs from our first experiment to include only those for `git` commands in this experiment, totaling 21 test inputs.

5.5. Results

Table 6 lists the results of this experiment. As can be observed, providing two additional examples (the 3-shot condition) improved performance. However, providing seven examples *reduced* performance.

Manual inspection of the synthesized code results reveals that examples in the 7-shot condition may be biasing the models towards unnecessary and/or incorrect arguments. Consider the following test input under the 7-shot condition for the `fetch` command:

```
# fetch changes from the server
```

Correct calls in the Faithful API would include: `fetch()`, `fetch(remote="")`, and `fetch(remote="origin")`. Per-API formulations of the former were used as the ground truth. (The latter two are semantic equivalents that require knowledge of `git`'s internal workings and default names to formulate, which a LLM is unlikely to have encoded.)

When this test language input is sent to the model, it produced the following output:

```
# In the Faithful API...
fetch(remote="server")

# In the Terse API...
f(r="server")

# In the Generic API...
check_for_updates(remote="server")

# In the Random API...
crackpot(organize="server")
```

In all cases, the PaLM model is generating a function call with the correct syntax, function names, and arguments names, but with an argument value of "server" when no argument value is required.

To understand why this may be occurring, we observe that the prompt is structured such that the first seven examples are calls to `fetch` without arguments (the ground truth formulation of a correct call). However, the last seven examples are calls to `fetch` with a string extracted from the natural language input. In fact, several of these inputs use a similar prefix as the test input (e.g., "# fetch changes from…"), though none are identical. Thus, it appears that these latter examples bias the model toward synthesizing a function call with an unnecessary argument.

6. Discussion

Summarizing the results from the experiments, we find that: 1) the model explicitly trained for code synthesis outperforms other models, suggesting the value of the training data for the task of synthesizing code for new APIs provided at inference time; 2) using random names (i.e., names outside the problem domain) is a surprisingly effective strategy for PaLM model variants; 3) most of the errors derive from models hallucinating code, and; 4) increasing the number of example uses of a function can increase the likelihood of the model using an API correctly (but perhaps to a limit).

These results suggest that LLMs can synthesize code for novel APIs defined at inference time, and that there are specific strategies one can employ to increase the likelihood of correct usage. One of the more interesting threads of future research is to examine additional API design alternatives to understand which API designs can be more effectively learned than others.

6.1. Limitations

The largest limitation of this work is the relatively narrow swath of the API design space explored. The APIs design space is large, meaning there are many design criteria that can be tested, such as: the inclusion of type information, the use of positional and named argument conventions, and the use of configuration dictionaries instead of named or positional arguments. We also did not test the effects of ordering prompt examples, which prior work (Zhao et al., 2021) and our own results suggest can be influential.

This research tested APIs for a single problem domain, for which we manually curated both the prompt examples and the test inputs. These factors may limit the generalizability of our results. For a task such as this to become a benchmark for other research, additional work should be done to craft a larger and more diverse corpus across multiple API surfaces, using methods such as crowd-sourcing (Y. Jiang et al., 2017; Krishna et al., 2017) or synthetic data generation (Wood-Doughty et al., 2021) to counteract the potential for implicit and unintended curator biases.

We did not compare the performance of these models against an API that is known to be in their training data. Future research should develop a method for identifying when a benchmark API design exists in a model's training data (whether intentionally or coincidentally), and compare model performance between these conditions. In addition to potential performance gains, controlled studies could provide useful insight into the effects of task-specific, task-adjacent, or task-agnostic natural language inputs on the model's ability to learn to use the API.

This work exclusively uses prompt engineering. Other work has shown that model fine-tuning (Wei et al., 2021) and prompt-tuning (Lester et al., 2021; Logan IV et al., 2021) can improve model output compared to prompt engineering. We would expect that a model fine-tuned on examples of an API would outperform prompt engineering with an un-tuned model, at the expense of (potentially per-API) training and serving costs. Although it is unclear how effective prompt-tuning would be for this specific use case, it offers some unique possibilities that make it a compelling area for future work. Specifically, prompt-tuning can be performed on a per-prompt basis, and requires only a single, fixed model. This allows one to dynamically load different APIs as needed, a particularly useful capability if two API designs overlap in the types of natural language phrases they accept (e.g., “add the .py files” has a particular meaning for `git`, and a different meaning for an FTP application).

7. Future Work

In this research, we used a relatively small set of measures to compare model performance. A promising area of future work is to employ salience methods (Bastings et al., 2021) or training data attribution (Pruthi et al., 2020) to determine which parts of the input affect model performance. Using these methods together with visual analysis tools (such as LIT; Tenney et al., 2020) could enable deeper insight into the relationships between prompt designs, inputs, and model outputs vis-a-vis API design strategies.

The models used in this research represent a modest sampling of current models. Obvious areas for future work include testing SynAPIs with models of different sizes and architectures. Size may be particularly approachable, as models like GPT-3 have readily available size class variants. In general, it would be useful to understand patterns of behavior as a function of architecture, size, and training regime.

Finally, should the research described above bear fruit, it would be prudent to assess the economic and business value of SynAPIs. This includes the cost of maintaining SynAPIs, the cost of operating the LLMs, and the capabilities and affordances of different system designs that employ SynAPIs.

8. Conclusion

This paper examined how well current LLMs can synthesize code in a novel API at inference time, and whether some API designs are easier for models to use than others. Our experiments demonstrate that modern LLMs can synthesize code for new APIs, and that choices in the API designs do affect usability. These results suggest that SynAPIs—APIs designed expressly for use by LLMs—are a worthy area of continued research, and could have larger implications for how LLMs are employed in human-AI systems.

References

Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., Finn, C., Gopalakrishnan, K., Hausman, K., Herzog, A., Et al. (2022). Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*.

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, *51*(4). https://doi.org/10.1145/3212695

Attanasio, G., Pastor, E., Di Bonaventura, C., & Nozza, D. (2022). Ferret: A framework for benchmarking explainers on transformers. *arXiv preprint arXiv:2208.01575*.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021). Program synthesis with large language models. arXiv. https://doi.org/10.48550/ARXIV.2108.07732

Bastings, J., Ebert, S., Zablotskaia, P., Sandholm, A., & Filippova, K. (2021). ” will you find these shortcuts?” a protocol for evaluating the faithfulness of input salience methods for text classification. *arXiv preprint arXiv:2111.07367*.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., … Amodei, D. (2020). Language models are few-shot learners (H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin, Eds.). In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems*, Curran Associates, Inc.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde de Oliveira Pinto, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). Evaluating Large Language Models Trained on Code. *arXiv e-prints*, arXiv 2107.03374, arXiv:2107.03374.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., … Fiedel, N. (2022). Palm: Scaling language modeling with pathways. arXiv. https://doi.org/10.48550/ARXIV.2204.02311

Flauzino, M., Verissimo, J., Terra, R., Cirilo, E., Durelli, V. H., & Durelli, R. S. (2018). Are you still smelling it? a comparative study between java and kotlin language, In *Proceedings of the vii brazilian symposium on software components, architectures, and reuse*.

Git reference manual [Accessed: 2022-06-01]. (2022).

Github copilot [Accessed: 2021-09-02]. (2021).

Jiang, E., Toh, E., Molina, A., Donsbach, A., Cai, C. J., & Terry, M. (2021). Genline and genform: Two tools for interacting with generative language models in a code editor, In *The adjunct publication of the 34th annual acm symposium on user interface software and technology*, Virtual Event, USA, Association for Computing Machinery. https://doi.org/10.1145/3474349.3480209

Jiang, E., Toh, E., Molina, A., Olson, K., Kayacik, C., Donsbach, A., Cai, C. J., & Terry, M. (2022). Discovering the syntax and strategies of natural language programming with generative language models, In *Chi conference on human factors in computing systems*, New Orleans, LA, USA, Association for Computing Machinery. https://doi.org/10.1145/3491102.3501870

Jiang, Y., Kummerfeld, J. K., & Lasecki, W. S. (2017). Understanding task design trade-offs in crowdsourced paraphrase collection. *arXiv preprint arXiv:1704.05753*.

Kaur, G., & Fuad, M. M. (2010). An evaluation of protocol buffer, In *Proceedings of the ieee southeastcon 2010 (southeastcon)*. IEEE.

Krishna, R., Zhu, Y., Groth, O., Johnson, J., Hata, K., Kravitz, J., Chen, S., Kalantidis, Y., Li, L.-J., Shamma, D. A., Et al. (2017). Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International journal of computer vision*, *123*(1), 32–73.

Lester, B., Al-Rfou, R., & Constant, N. (2021). The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., d'Autume, C. d. M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., … Vinyals, O. (2022). Competition-level code generation with alphacode. arXiv. https://doi.org/10.48550/ARXIV.2203.07814

Logan IV, R. L., Balažević, I., Wallace, E., Petroni, F., Singh, S., & Riedel, S. (2021). Cutting down on prompts and parameters: Simple few-shot learning with language models. *arXiv preprint arXiv:2106.13353*.

Perez De Rosso, S., & Jackson, D. (2013). What's wrong with git? a conceptual design analysis, In *Proceedings of the 2013 acm international symposium on new ideas, new paradigms, and reflections on programming & software*.

Pruthi, G., Liu, F., Kale, S., & Sundararajan, M. (2020). Estimating training data influence by tracing gradient descent. *Advances in Neural Information Processing Systems*, *33*, 19920–19930.

Tenney, I., Wexler, J., Bastings, J., Bolukbasi, T., Coenen, A., Gehrmann, S., Jiang, E., Pushkarna, M., Radebaugh, C., Reif, E., Et al. (2020). The language interpretability tool: Extensible, interactive visualizations and analysis for nlp models. *arXiv preprint arXiv:2008.05122*.

Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., Et al. (2022). Lamda: Language models for dialog applications. *ArXiv preprint*, *abs/2201.08239*. https://arxiv.org/abs/2201.08239

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., & Le, Q. V. (2021). Finetuned language models are zero-shot learners. arXiv. https://doi.org/10.48550/ARXIV.2109.01652

Wood-Doughty, Z., Shpitser, I., & Dredze, M. (2021). Generating synthetic text data to evaluate causal inference methods. *arXiv preprint arXiv:2102.05638*.

Wu, Y., Rabe, M. N., Hutchins, D., & Szegedy, C. (2022). Memorizing transformers, In *International conference on learning representations*. https://openreview.net/forum?id=TrjbxzRcnf-

Zhao, Z., Wallace, E., Feng, S., Klein, D., & Singh, S. (2021). Calibrate before use: Improving few-shot performance of language models, In *International conference on machine learning*. PMLR.