Measuring Novel API Use in Program Synthesis

Ethan A. Chi[1 2*], **Yuhuai Wu**[1 2], **Aitor Lewkowycz**[1], **Vedant Misra**[1]
Anders Andreassen[1], **Vinay Ramasesh**[1], **Ambrose Slone**[1], **Guy Gur-Ari**[1],
Behnam Neyshabur[1], **Ethan Dyer**[1]

[1] Google Research, Blueshift Team
[2] Department of Computer Science, Stanford University

Abstract

Large Language Models (LLMs) have demonstrated impressive program synthesis abilities, with high performance on benchmarks of real-world programming problems suggesting a strong fundamental ability to reason. However, a systematic analysis of the extent of these abilities can be challenging, due to performance being confounded by memorization. In this work, we propose synthetically generating *API use programming problems*, which require reading and working with novel APIs. Abstracting away complex, memorized operations through these APIs—which are defined solely within the scope of the problem—allows us to examine the ability of LLMs to perform tool use in a controlled setting while avoiding spuriously high performance from memorization. Our problem generation process is highly configurable, enabling us to examine the relative influence of components of problem structure (e.g. compositionality) on LLM reasoning performance in a controlled way. We create benchmarks examining some of the most relevant such components, against which we evaluate standard program synthesis models.

1 Introduction

Large language models (LLMs) have recently achieved success on synthesizing general-purpose languages such as Python or C++ [Chen et al., 2021, Nijkamp et al., 2022]. Trained on large quantities of text including real-life code samples, such models can solve a large variety of often-complex prompts without explicit training [Austin et al., 2021]—including substantially complex problems considered challenging for human programmers [Hendrycks et al., 2021]. This has resulted in rapid deployment in the wild (e.g. by Github Copilot; Chen et al. [2021]).

In addition to its obvious engineering applications, the task of code synthesis offers an exciting opportunity to study large LLMs' ability to reason [Wei et al., 2022, Nye et al., 2021]. Solving complex coding questions at a human-like level requires a variety of skills—compositionality, memorization, long-term reasoning, and tool use, to name a few [Chen et al., 2021, Li et al., 2022]. Thus, an analysis of what elements of code synthesis are within the capabilities of state-of-the-art models—and which remain challenging—might shed light on current LLMs' reasoning abilities.

Unfortunately, such analysis is confounded by *memorization*. On one hand, common "coding interview"-type questions, which one might expect to require extensive compositional reasoning, are surprisingly easy to solve for language models [Chen et al., 2021]. On the other hand, LLMs can often struggle given a less common but perfectly reasonably formulated question, instead providing answers to a more common prompt (Figure 1). Taken together, these cases encourage us to ask: to

[*] Work performed while an intern at Google Research, Blueshift Team.

36th Conference on Neural Information Processing Systems (NeurIPS 2022) Track on Datasets and Benchmarks.

Prompt

```
Write a python function to identify non-prime numbers.
```

Generated

```
def is_prime(num):
    for i in range(2, num):
        if num % i == 0:
            return False
    return True
```

Figure 1: A common failure mode for program synthesis: distraction by a more common memorized problem. Prompted to write a function which identifies non-prime numbers, OpenAI Codex [Chen et al., 2021] generates a solution to a more common problem: identifying prime numbers, i.e. the Boolean inverse of the prompted problem. (This decode was sampled 1-shot with temperature 0.5 from `code-davinci-002`.)

Consider the following API:

`image.rotate()`	Returns the image rotated by the given number of degrees.
`image.distort(pixels)`	Returns the image distorted by the given number of pixels.
`image.get_width()`	Returns the width of the image in pixels.

Prompt:

```
Write a function that distorts an image by a number of pixels equal to its width.
[BEGIN]
def distort_image_by_width(image):
```

Solution: `image.rotate(pixels=image.get_width())`

Figure 2: An (abridged) example of a synthetic library and one possible programming problem based around that library.

what extent does performing well on current benchmarks actually reflect strong reasoning abilities, rather than simply improvements in retrieving code from memory?[2]

In this work, we propose to sidestep these questions through by evaluating models on *synthetic* programming problems. Each of our generated problems relies upon **synthetic libraries**: end-user-defined Python libraries with no actual code definitions, merely natural-language documentation. We present these APIs in the prompt itself, framed as function definitions in the style of any standard Python library (Figure 2).

By delegating to such black-box libraries, we avoid forcing the LM to generate complex but standard algorithmic steps (e.g. a prime search or factorial computation). In addition to making evaluation significantly easier, this ideally also avoids biasing our evaluation towards whether the LM can generate these steps; in theory, no further knowledge beyond the fundamental ability to generate code should be required to solve our generated problems.

More broadly, synthetically generating problems allows us to tease out the various components of reasoning: by adjusting the parameters, one can build a problem that requires extensive compositional reasoning without significant memorization, or tool use without [whatever we even call argument fixing]. By sweeping over each of these dimensions, one can test the impact of different components of reasoning in code synthesis (Section 4).

Specifically, our contributions are as follows:

- A set of **synthetic libraries**, plausible Python libraries that each contain function signatures accompanied by natural-language definitions. In addition to several synthetic libraries provided in our package, benchmark users can also easily define their own libraries, either programmatically or declaratively.

[2]In fact, existing benchmarks have a strong bias towards testing raw memorization: taking the Mostly Basic Programming Problems (MBPP) dataset [Austin et al., 2021] as an example, 30% of a sample of 100 questions require knowledge of an external definition to solve[3].

- **A Python package**, `api_use`, which generates synthetic programming problems centered around applying synthetic libraries. Each programming problem first presents documentation for functions from one or more synthetic libraries, then gives natural-language instructions for the function to be written (Figure 2). Rather than generating accompanying test cases, we simply evaluate the flow of operations against an automatically generated reference solution. This avoids the challenges of achieving sufficient coverage over test cases to ensure correctness.
- The ability to **configure** a large number of components of our programming problems, and **a series of benchmarks** generated by sweeping over some of the most significant components. Model performance is Validating two different sizes of Code [Austin et al., 2021] against our generated benchmarks, we find that models mostly do well but are vulnerable to a number of different factors that are in-domain for program synthesis.

We view these benchmarks as orthogonal to existing code generation datasets; the tasks generated via our approach cannot possibly be representative of even tool use in particular, much less the full complexity of the distribution of real-life coding problems, due to their synthetic nature. Rather, we aim to examine the impact of certain factors of problems in a *controlled* yet reasonably realistic setting. Our approach thus follows in the vein of the *challenge sets* of NLI, as introduced by McCoy and Linzen [2018]: specific in-domain examples which allow us to examine the full spectrum of model behavior in a controlled way. Overall, we frame our approach as a general methodology for better understanding the abilities of program synthesis models through such targeted analysis. Towards this end, we release our package, demo notebook, and accompanying benchmarks[4].

2 Synthetic Libraries

For the purpose of constructing API use problems, we define the concept of a **synthetic library** (or synthetic API). These libraries do not actually exist as fleshed-out Python code: rather, they consist solely of function signatures accompanied by natural-language definitions. Importantly, the function definitions are typed internally; this allows our package to compute appropriate references for chained, nested, or composed function calls.

We provide a set of synthetic libraries (Table 1), upon which we construct our benchmarks (Section 5). In addition to these provided API's, benchmark end users can also define their own API's. This can be done either declaratively (through JSON) or programmatically (through a Python interface).

Library	Task	Example call
`solids`	Computing attributes of n-D solids	`solids.get_volume_of_cylinder(5, 2, 3)`
`image`	Manipulating images	`image.rotate(45).blur()`
`chem`	Manipulating and searching through molecules	`molecule.get_atom_with_symbol("F").get_label()`
`music`	Searching through songs and melodies	`melody.get_note_with_pitch("A").to_midi()`

Table 1: A list of synthetic libraries provided in the `api_use` package. Additional libraries can also be easily defined by an end user of the benchmark.

Library types We support two related but similar kinds of synthetic libraries, corresponding to two major kinds of API's commonly found in Python code. **Importable** libraries are typically imported at the top level, providing global-level functions (e.g. `random` or `re`). **Class** libraries are used as the instance of a class, providing member functions which operate on some object. As the underlying code supporting these libraries is identical, objects from each kind of library can be chained with each other (Figure 3).

[4] `http://ethanachi.com/api_use`.

```python
# importable library
import solids
solids.get_volume_of_sphere(radius=5)
```

```python
# class library
image = Image()
image.flip_horizontal().blur()
```

Figure 3: Importable vs. class libraries.

3 Synthetic Problems

With a set of synthetic libraries defined, a synthetic API use problem can be generated from a *function signature*: a single function call—or composition of function calls—using functions from those synthetic libraries. The goal of the resulting problem is to produce code which replicates this signature. The problem's prompt first provides documentation for the synthetic libraries: at a minimum, all the functions used in the target function signature must be included, in addition to a configurable number of distractors. The prompt then provides a natural language description of the function to be written (e.g. *Write a function that gets the volume of a cone*). Successfully solving the problem thus entails reading the definitions, identifying the function(s) to be used, and composing them appropriately to match the natural-language prompt (Figure 4).

signature `solids.volume_of_cone()`
description `'gets the volume of a cone'`

Prompt:

```
Consider the following functions:
def solids.volume_of_cone(radius, height):
    Calculates the volume of a cone with the given radius and the given height.

Write a function that gets the volume of a cone.
[BEGIN]
import solids
def test(radius, height):
```

Figure 4: A minimal example of a synthetic API use programming problem.

Evaluation To evaluate a proposed solution, we parse the solution as a Python function, substituting the non-existent API (e.g. `solids`) with a dummy object which records any methods called on it. We then compare the data flow of the method calls against the original function signature itself, marking the solution as correct if the data flow matches. This approach ensures correctness of the overall computational graph while enabling modifications to the exact order in which code is evaluated. In addition, it avoids requiring test cases altogether, which can often fail to pick up subtle differences with respect to the original definition [Austin et al., 2021].

4 Controlling Generation: Disentangling the Factors of Reasoning

One major advantage of synthetically generating problems is *controllability*: an end user can tightly control all characteristics of a generated problem. Examining programming problems drawn from real-life sources, even problems that use similar skills or forms of reasoning may differ significantly in their use of memorization, involvement of other form of reasoning, or a generally different problem structure. The lack of a *minimal pair* between problems thus makes it difficult to make a strong causal statement about the interaction between factors of reasoning and model performance. By contrast, our synthetic API use problems are designed to allow for the easy creation of minimal pairs. Our package exposes a large number of parameters, most of which correspond to some factor of the reasoning process. Therefore, by modifying one or more of these parameters, one can create a minimal pair, benchmarking a model against two variants of the same problem and understanding the causal influence of problem characteristics on model performance.

4.1 Composition

Requiring models to use multiple functions and connect their results to each other tests their ability to perform composition, a core part of reasoning. Towards this end, arbitrarily compositional function signatures can be transformed into programming problems. The results of one function can be passed both as an argument to another function (Figure 5a) and as the subject of another function call (Figure 5b). In this second case, the return type of the first function call is used to disambiguate the library for the next function call. For example, in the below example the return type of `get_atomic_with_atomic_num` in library molecule is defined to be `atom`, so the second function call is disambiguated to `atom.get_atomic_weight()`. Adjusting the compositionality of the generated problem allows us to understand the impact of both the length and width of the tree of function calls (Figure 5).

Prompt:

```
Write a function that rotates an image by a number of pixels equal to its width.
[BEGIN]
def rotate_image_by_width(image):
```

Solution: `image.rotate(pixels=image.get_width())`

Prompt:

```
Write a function that flips an image horizontally, then rotates it by the given number of degrees,
      then distorts it by the width of the image, then blurs it by the given number of pixels.
[BEGIN]
def flip_horizontal_then_rotate_then_distort_by_width_then_blur(image, degrees, pixels):
```

Solution: `return image.flip_horizontal().rotate(degrees)`
`.distort(image.get_width()).blur(pixels)`

Figure 5: Sample generated problems testing LLM ability to perform composition. Distractors omitted for brevity.

4.2 Synthetic API documentation

A first prerequisite to successfully solving a generated API use problem is to identify the functions listed in the documentation necessary to solve the problem. In this section, we examine generation options for the documentation itself. By controlling these options, one can examine the impact to which documentation style confounds performance on API use.

Distractor functions By default, only those functions called as part of the signature are added to the documentation block; however, this makes selecting the correct function to be used trivial. To make the function-selection task more challenging, an arbitrarily large number of distractor functions can be added. These distractor functions are randomly sampled from the corresponding synthetic libraries from which the original functions called are drawn. The position of the target function within these distractors can also be customized, in order to probe for the existence of either recency or primacy bias (Deese and Kaufman [1957]).

Function, argument names From a human perspective, one might expect the synthesized code to be dependent on its natural-language description (e.g. *write a function which computes the volume of a sphere*). Yet there is some evidence that this isn't the only thing that affects model outputs: for example, when test cases are provided, Austin et al. [2021] demonstrate that LLM's may ignore the description altogether, basing their behavior on the contents of the test cases. Generalizing this to the documentation A natural question in the context of the API use benchmark, then, is to what extent does picking the right function depend on the function name as opposed to the natural language documentation? Towards this end, the `api_use` package offers several options for **noising** either of these names; the function names can be set to be randomly shuffled, sequential, or completely randomly generated, while the documentation can be randomly shuffled or even omitted.

Formatting By default, the synthetic API function list is formatted similarly to a standard Python library (Figure 4). However, this formatting can be customized arbitrarily through a formatting callback, e.g. to produce natural-language-style documentation, or even to present the functions in plain English.

4.3 Argument handling

Synthesizing full-featured programs necessarily requires the ability to work with *subroutines*, or dependent function calls. Passing data to such subroutines requires a flexible handling of data flow: not all data in the global scope will be passed to a subroutine, while additional arguments may need to be computed. Those arguments that are passed may be in a different order from the global scope. Successfully working with subroutines, then, requires learning an equivalence between *outer* function arguments—the arguments to the outer function to be synthesized—and *inner* function arguments—the arguments to the subroutine itself.

We view the process of calling and working with synthetic API's as an effective testing ground for evaluating LLM understanding of this equivalence. Our generated programming problems expose multiple ways to modify the standard outer-inner argument correspondence. Specifically, our package supports the following options; each option can be configured through simple modifications to the function signature (Figure 6).

Fixed arguments Any arguments to a synthetic API call can be *fixed*, or set to a constant value. For example, given a synthetic API function `solids.volume_of_cone(height, radius)`, one could create a problem which tests the ability of a LLM to compute the volume of a cone with a known height but unknown radius. Solving the resulting programming problem requires a LLM to understand which API arguments are equivalent to the outer arguments, and which are being set to fixed values (Figure 6).

Argument ordering The *outer* arguments—the arguments to the function to be synthesized—must correspond to all unfixed arguments to API calls within the function. However, these outer arguments can be in any order— for example, given a synthetic API function `solids.volume_of_cone(height, radius)`, one could create a problem which tests the ability of a LLM to compute the volume of a cone given its radius and height, i.e. in the opposite order from the API— and be named differently. Solving the resulting programming problem requires a LLM to understand which API arguments are equivalent to the outer arguments.

Prompt:

```
Write a function that gets the volume of a
      cone with radius 3.
[BEGIN]
import solids
def test(height):
```

Solution: `return solids.volume_of_cone(3, height)`

Prompt:

```
Write a function that gets the volume of a
      cone with radius a and height b.
[BEGIN]
import solids
def test(a, b):
```

Solution: `return solids.volume_of_cone(a, b)`

Figure 6: Modifying the argument order or names when prompting the model allows us to testing LLM understanding of the invariance of argument handling between outer and inner calls.

5 Benchmarks

5.1 Motivation

By modifying the configuration parameters described in Section 4, one can generate multiple variations of a problem along a particular axis of reasoning. Compiling many such variations of the same problem produces a **benchmark** which correlates LM performance against this axis. The synthetic nature of the problem generation process means that the pairs are as minimal as possible, allowing an unbiased evaluation of the impact of various factors of reasoning.

The `api_use` package allows for the easy creation and evaluation of these kinds of benchmarks, which we represent as a JSON files enumerating the parameters of each problem. Creating a new benchmark simply entails writing a script which sweeps over one or more parameters of interest.

5.2 Provided Benchmarks

We provide a set of pre-built benchmarks which sweep over several significant axes of reasoning. Each benchmark iterates through the Cartesian product of the factors under modification.

COMPOSITIONALITY Can models effectively compose functions, using the output of one function call as the input to the next? To what extent does the length of this chain affect generation correctness? This benchmark evaluates model performance on performing a series of image manipulation operations on a single image. Each operation takes in an intermediate stage of the image and produces a new image; thus, the overall structure is that of a series of chained function calls. Varying the length of this chain allows one to examine LLM performance as a function of compositional depth.

DISTRACTORS When presented with a large number of functions from the same API, can models still select the correct function to complete a task? This benchmark evaluates model performance on solving a single-function-call problem (calculating the volume of a cylinder) with respect to both the number of distractors (up to 8) and the position of the target function (anywhere from the beginning to the end of the function documentation).

SEMANTICS Our programming problems document synthetic APIs as a set of $\{$function name, natural language description$\}$ pairs. In a typical real-world library, both of these are descriptive of the function to be written. Do models choose which functions to use based on the function name, the function description, or both? Similarly, the prompt itself has both a function name and a description: to what extent does the model rely on either of these to make a decision as to which function it calls? This benchmark evaluates model performance on solving a single-function-call problem (calculating the volume of a cone) under both non-semantic conditions (function names/descriptions set to sequential numbers, e.g. `func1`) and adversarial conditions (function names/descriptions completely scrambled). These conditions are evaluated for both the documentation and the prompt itself. We omit completely unsolvable combinations (e.g. both function name and description set to adversarial).

ARGUMENTS Can models flexibly work with argument order, passing arguments to an API in a different order or with a different degree of specification compared to the input? This benchmark evaluates model performance on solving a single-function-call problem—computing the volume of a cone—given various modifications to the arguments. Specifically, we examine every possible combination of arguments—fixed and unfixed, with the unfixed arguments in each possible order.

Few-shot prompting Previous work [Chen et al., 2021] has demonstrated that models do better when few-shot prompting is provided. In addition to zero-shot prompting, we test the following few-shot conditions when appropriate.

- Zero-shot prompting—as described above
- In-domain prompting (1-shot and 2-shot): all few-shot examples have exactly the same configuration other than function name as the target function signature.
- Out-of-domain (1-shot and 2-shot): all few-shot examples draw from the same library, but have a different configuration.
- Mixed IID/OOD (2-shot): one few-shot example has the same configuration as the target function signature, while one has a different configuration.

Each few-shot prompt entails a full prompt + function call: however, to be more realistic, the list of function definitions is shared between all few-shot prompts and the actual target, placed at the beginning of the prompt. In all cases, functions used in few-shot prompts are always included in the documentation and count against the number of distractors.

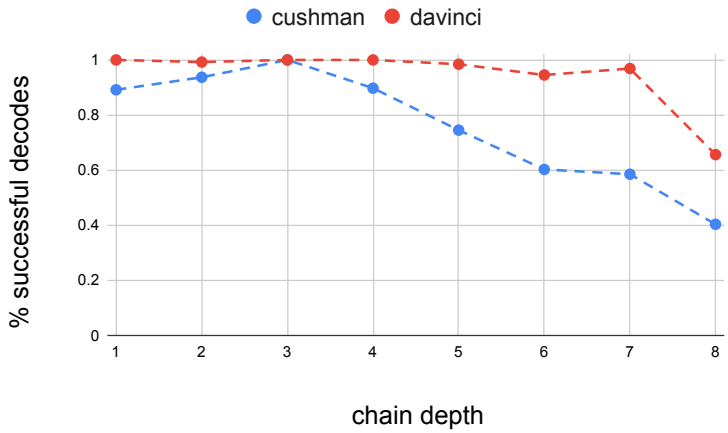

Figure 7: Codex performance on the COMPOSITIONALITY benchmark, which examines the ability of LLMs to generate increasingly long sequences of chained operations. Model performance, as measured by the fraction of correct decodes, drops significantly after several chains, with `cushman`, a smaller model, dropping off earlier than `davinci`.

	% correct			% correct	
Documentation noise	`cushman`	`davinci`	**Prompt noise**	`cushman`	`davinci`
Baseline	1.000	1.000	Baseline	1.000	1.000
Noise description(s):					
Empty	1.000	1.000	Empty	0.891	1.000
Scrambled	1.000	1.000	Scrambled	0.759	0.846
Noise function name(s):					
Numbered (`funcK`)	0.497	0.996	Meaningless (`func`)	0.997	1.000
Scrambled	0.000	0.000	Adversarial	0.998	1.000

Table 2: LLM accuracy on the SEMANTICS benchmark, which tests a model's ability to identify the correct function to be used given noise to function name or documentation. We examine the impact of noising both the list of functions (which defines the synthetic API) and the actual prompt itself.

5.3 Evaluation

To understand the performance of state-of-the-art program synthesis models on synthetic API use problems, we evaluate two sizes of OpenAI Codex—`code-davinci-002` and `code-cushman-001` [Chen et al., 2021] —on our benchmarks. For each model, we decode 128 samples with temperature 0.5. As a metric, we compute the fraction of decodes which successfully solve the problem.[5]

5.4 Results

DISTRACTORS Regardless of the number of distractors and their placement, both `cushman` and `davinci` achieve perfect performance (100% accuracy on decodes). For other benchmarks, we thus always place the target function in the middle of the documentation and use a constant, nonzero number of distractors based on the particular synthetic library being used.

COMPOSITIONALITY As one might reasonably expect, model performance (as measured by the percentage of successful decodes) decreases with increasing compositional depth (Figure 7). Performance begins to fall off for `cushman` around 4 chains and for `davinci` around 8 chains. This suggests that even longer chains, or alternatively wider chains, could demonstrate further vulnerabilities in the performance of larger program synthesis models such as `davinci`.

[5]We use the OpenAI API to sample all decodes and provide a script to evaluate in our repository.

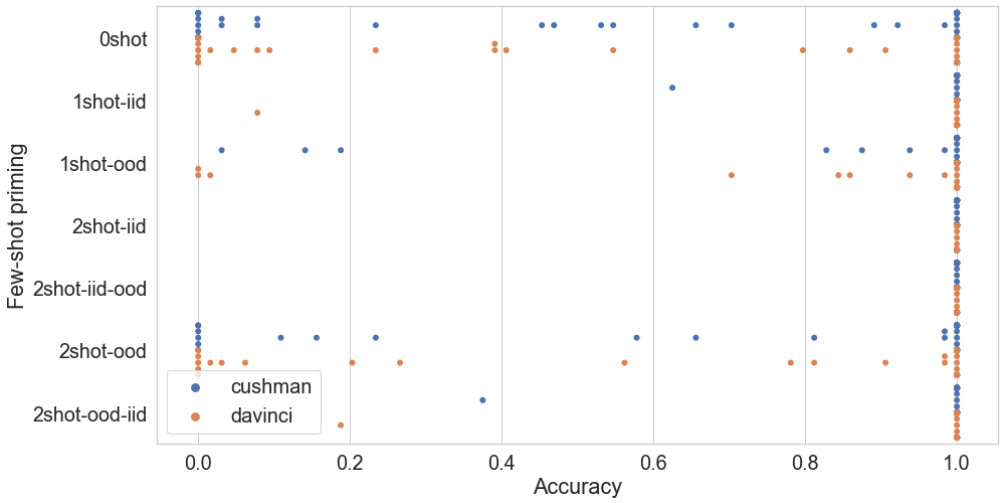

Figure 8: Model performance on the ARGUMENTS benchmark, 4-argument case. Most problems are solved with perfect accuracy (large clusters around 1.0); however, in the non-IID fewshot priming cases, some argument orders tend to be significantly worse.

SEMANTICS Looking at documentation (Table 2), models achieve perfect performance when the function descriptions are empty or scrambled; however, when the function names are numbered, performance immediately drops for `cushman` (albeit not `davinci`), and models are entirely unable to solve problems where the function names have been scrambled. This suggests that Codex prioritizes making decisions on function retrieval based on function names, not documentation. On the other hand, when reading the prompt itself, Codex does well on both conditions; a non-descriptive or adversarial function name does not impact performance, while providing an adversarial description and descriptive function name still does reasonably well.

ARGUMENTS Almost all of the problems in this benchmark are solved by models with perfect accuracy over all decodes sampled. However, there are distinct exceptions, especially without in-domain priming. With out-of-domain priming (i.e. with exactly the same library and style of function calls, simply with a different argument order), several argument orders prove to be extremely challenging for LLMs despite being perfectly reasonable ways of calling functions. For example, passing arguments in the order $2, 4, 1$ and fixing the 3rd argument has only 0.031 of successful decodes on `davinci`, and no successful decodes on `cushman`.

6 Discussion

In this work, we have presented a methodology for generating a tool use benchmark based around the concept of applying synthetic APIs. By generating tightly controlled benchmarks, the impact of a large variety of factors can be examined on the ability of LLMs to reason. Additionally, providing logical primitives through synthetic libraries allows us to evaluate programming problem performance separately from simple memorization, allowing for a more rigorous understanding of the reasoning capabilities of state-of-the-art program synthesis models.

We do not view our methodology as replacing, but rather complementary to, existing programming problem benchmarks. Rather than attempting to evaluate the complete diversity of programming tasks that exist in the wild, our approach allows for the creation of minimal pairs and challenge sets that enable one to hone in on the exact factors that make a problem solvable or unsolvable. Our flexible approach to generating synthetic problems could be adapted to test a wide variety of hypotheses.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language

models. *arXiv preprint arXiv:2108.07732*, 2021.

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

James Deese and Roger A Kaufman. Serial effects in recall of unorganized and sequentially organized verbal material. *Journal of experimental psychology*, 54(3):180, 1957.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode, 2022. URL `https://arxiv.org/abs/2203.07814`.

R Thomas McCoy and Tal Linzen. Non-entailed subsequences as a challenge for natural language inference. *arXiv preprint arXiv:1811.12112*, 2018.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.