This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3267446



# MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, Abhinav Jangda§

**Abstract**—Large language models have demonstrated the ability to generate both natural language and programming language text. Although contemporary code generation models are trained on corpora with several programming languages, they are tested using benchmarks that are typically monolingual. The most widely used code generation benchmarks only target Python, so there is little quantitative evidence of how code generation models perform on other programming languages. We propose MultiPL-E, a system for translating unit test-driven code generation benchmarks to new languages. We create the first massively multilingual code generation benchmark by using MultiPL-E to translate two popular Python code generation benchmarks to 18 additional programming languages.

We use MultiPL-E to extend the HumanEval benchmark [1] and MBPP benchmark [2] to 18 languages that encompass a range of programming paradigms and popularity. Using these new parallel benchmarks, we evaluate the multi-language performance of three state-of-the-art code generation models: Codex [1], CodeGen [3] and InCoder [4]. We find that Codex matches or even exceeds its performance on Python for several other languages. The range of programming languages represented in MultiPL-E allow us to explore the impact of language frequency and language features on model performance. Finally, the MultiPL-E approach of compiling code generation benchmarks to new programming languages is both scalable and extensible, making it straightforward to evaluate new models, benchmarks, and languages.

## 1 Introduction

*Code generation models*, also known as large language models (LLMs) of code, are deep neural networks trained on massive corpora of source code. Over the past few years, code generation models have demonstrated their utility on a wide variety of software engineering tasks, including test generation, documentation generation, and even synthesizing working programs from natural language descriptions [1, 4, 5, 6, 7]. New products such as GitHub Copilot[1], Amazon CodeWhisperer[2], and Tabnine[3] built on code generation models are growing in popularity with developers [8]. Although several code generation models are trained on multiple programming languages, they are typically only evaluated on a single programming language: Python. Machine learning researchers are familiar with Python: they have painstakingly constructed several Python code generation benchmarks [1, 2, 9, 10] and it is the best represented language in training datasets [1, 2, 10, 11]. However, we should also evaluate code generation models with other languages to support a wider variety of programmers. There is prior work on multi-language evaluation [7], but it uses perplexity as a proxy for performance, instead of benchmarks that check correctness.

In this paper we present MultiPL-E, a system for translating code generation benchmarks from Python into new languages, and use it to propose the first massively parallel, multi-language benchmark for code generation. By "multi-language" we mean multiple programming languages: MultiPL-E supports 18 languages and is straightforward to extend with more. By "parallel", we mean that MultiPL-E produces parallel problems for each language, thus we can measure performance of a code generation model on a consistent set of problems across multiple programming languages. What makes MultiPL-E possible is that code generation benchmarks have unit tests to determine if the generated function behaves correctly.

MultiPL-E uses a suite of 18 compilers from Python benchmarks to each target language.[4] However, what makes this scale is that these are *not* full-fledged compilers. Each compiler must be able to translate four components from Python: (1) a function signature (name and arguments), (2) simple unit tests, (3) a comment describing the expected function behavior, and (4) type annotations if the target language is statically typed. Notably, the compiler does not have to translate the body of a function, since it is the job of the code generation model to synthesize it. Thus each MultiPL-E compiler is approximately 200 LOC and easy to build. MultiPL-E also includes a simple, rule-based tool to translate technical terms in comments to be more language appropriate, e.g. a Python list is approximately a C++ vector.

MultiPL-E also includes a containerized sandbox that (1) compiles programs if necessary, (2) runs them with appropriate timeouts, (3) validates their results on unit tests, and (4) classifies each output as successful, syntax error, etc. Thus each language requires an evaluation script, which is typically about 20 LOC.

We use MultiPL-E to translate two widely-used code generation benchmarks, HumanEval [1] and MBPP [2], into

§. Authors are listed alphabetically with students first, then faculty.

1. https://github.com/features/copilot/
2. https://aws.amazon.com/codewhisperer/
3. https://www.tabnine.com/

4. These source-to-source compilers are sometimes called *transpilers*.

18 languages. The 18 languages capture a broad spectrum of language features, application areas, and popularity, allowing us to explore the impact of these factors on model performance.

We use the multi-language parallel MultiPL-HumanEval and MultiPL-MBPP benchmarks to evaluate three state-of-the-art code generation models: Codex [1], CodeGen [3], and InCoder [4]. Our evaluation presents new insights into the effectiveness of code generation models, including:

1) Across models and benchmarks, code generation models perform extremely well on JavaScript, sometimes outperforming Python, even on benchmarks originally designed to evaluate Python performance. Codex also performs well on C++, Scala, and TypeScript.
2) There is no strong correlation between model perplexity and correctness of generated code, which suggests that perplexity may not be a good estimate of performance.
3) Code generation performance is correlated with language popularity, but some niche languages perform as well as more popular languages.
4) Code generation performance is sensitive to prompt design for both niche and popular languages.
5) Static type-checking neither helps nor hinders code generation model performance.

To summarize, our key contributions are:

- MultiPL-E: a suite of compilers and an evaluation framework for translating code generation benchmarks from Python into other programming languages. MultiPL-E translates unit tests, doctests, Python-specific terminology, and type annotations.
- Two parallel benchmarks for code generation in 19 languages encompassing a variety of programming paradigms, language features, and popularity levels.
- A multi-language parallel evaluation of three models, Codex [1], InCoder [4], and CodeGen [3].
- Explorations of language frequency effects, the impact of type annotations, and prompt translation sensitivity on code generation performance, along with a fine-grained error analysis for four languages.

We hope this evaluation work will help the many software engineers that do not use Python to assess the feasibility of code generation models for their work and to understand the language factors that may affect model performance.

MultiPL-E is presently being employed to evaluate code generation models for the BigCode project, which is an open collaborative effort to build code generation models on permissively licensed code.[5]

The MultiPL-E system, dataset, and tutorial are available at github.com/nuprl/MultiPL-E.

## 2 Code Generation

Code generation has long been a task of interest: there is extensive work on program synthesis [12, 13, 14] using both symbolic and neuro-symbolic approaches. More recently, large neural language models (LLMs) trained for text generation have demonstrated the ability to perform program completion [5, 15, 16]. Since 2020, several code generation models have been trained or fine-tuned on programming language text [1, 4, 6, 7, 17, 18]. Unlike traditional program synthesis techniques, neural language models are able to condition on and generate both natural language (i.e., code comments) and programming language text. However, existing code generation models are tested using monolingual benchmarks that largely target Python. Thus there is little quantitative data about how well they perform on other languages. We make progress towards answering this question by proposing two large-scale parallel benchmarks for code generation in 19 languages, which we use to evaluate three state-of-the-art models: Codex, CodeGen, and InCoder.

5. bigcode-project.org

(a) The prompt given to the NL2Code model:

```
def lsi(lst):
    """Create a function that returns a
    tuple (a, b), where 'a' is the largest
    of negative integers, and 'b' is the
    smallest of positive integers in a
    list. If there is no negative or
    positive integers, return them as
    None.
    Examples:
      lsi([2, 4, 1, 3, 5, 7]) == (None, 1)
      lsi([]) == (None, None)
      lsi([0]) == (None, None)
    """
```

(b) A completion generated by Codex:

```
  if len(lst) == 0:
    return (None, None)
  else:
    neg_nums = [x for x in lst if x<0]
    pos_nums = [x for x in lst if x>0]
  if len(neg_nums) == 0:
    neg_nums = None
  if len(pos_nums) == 0:
    pos_nums = None
  return (max(neg_nums), min(pos_nums))
```

(c) 3 of the 12 unit tests (the generated solution fails two):

```
X assert lsi([2, 4, 1, 3, 5, 7]) == (None, 1)
X assert lsi([2, 4, 1, 3, 7, 0]) == (None, 1)
✓ assert lsi([1, 3, 4, 5, 6, -2]) == (-2, 1)
```

Figure 1: Problem 136 of 164 of the HumanEval benchmark. We shorten the name `largest_smallest_integers` for brevity. Top: the prompt for the model, with the function signature, natural language description, and doctests. Middle: a Codex-generated solution. Bottom: unit tests.

### 2.1 The Natural Language to Code Task

Code generation models have been applied to a variety of tasks, including test generation [19], docstring generation [20], code search [17, 21], type inference [22, 23, 24], and more [25]. We focus on the **natural-language-to-code** task (NL2Code): given the description of a function in natural language, complete the function body.

The input to a code generation model is called a *prompt*. Figure 1a shows an example prompt from the HumanEval

benchmark for NL2Code [1]. The prompt has several sources of information for the model: the function signature (its name and parameters); a brief comment describing the function; and, optionally, examples in the form of Python doctests. Given the prompt as input, the code generation model generates a *completion* that is likely to follow the given prompt.

Note that the model does not receive an explicit cue about the target language, but each of the three prompt regions provide implicit cues: the syntax of the function signature, the terminology used in the natural language description, and the syntax of the doctests all suggest that the target is Python. Consequently, to translate this prompt to a new programming language, we must target all three regions of the prompt.

### 2.2 Sampling Program Completions

There are several ways to configure how a code generation model produces completions, each of which can have a significant effect on the quality of generated code. Fundamentally, a completion is a sequence of tokens and is *not* an abstract syntax tree. Therefore, a completion can readily produce tokens that go beyond a single function. For example, given just the the signature of "mean", InCoder produces the mean, variance, standard deviation, and several other functions (Figure 2). In fact, it can continue producing code up to the maximum sequence length, which, for InCoder, is 2048 tokens.

We control this output by specifying *stop sequences* that typically demarcate the end of a function. For Python, we use the stop sequences that have been employed in prior work [1]. For example, when completing a top-level function, `\ndef` marks the start of the next top-level function, but allows nested helper functions. For other languages, we design different sets of stop sequences (§A).

Under the hood, given a prompt, a code generation model produces a completion one token at a time. At each step, the neural network receives an encoded prompt as input and produces a distribution for the following token. To generate several tokens, a *sampling algorithm* iteratively samples next tokens, extending the prompt at each step with the previously sampled token.

There are a variety of sampling approaches that one can use. A naive approach is to greedily sample the next most likely token, but this performs poorly in practice [26]. One approach employed in prior work [1], and in this article, is to rescale the probability distribution to favor high probability tokens more strongly using a *temperature* hyperparameter ($0 \leq t < 1$): low temperature makes the completion more "predictable" and high temperature makes it more "creative". This is commonly combined with top-$p$ sampling, which cuts off the least likely tokens that contribute in aggregate $1 - p$ to the probability mass, and redistributes their mass to the remaining tokens. Since these procedure is nondeterministic, we sample 200 completions for every prompt and choice of hyperparameters.

### 2.3 Evaluating Code Generation

Early work on code generation relied on textual similarity metrics for evaluation [17, 27]. However, previous work shows that textual similarity is not reliably correlated with code correctness [1, 2]. The best way to evaluate code generation is to test code correctness using a suite of hidden unit tests.

```
def mean(n):
    return sum(n)/len(n)

def variance(n):
    mean = mean(n)
    return sum([(n-mean)**2 for n in n])/len(n)

def standard_deviation(n):
    return math.sqrt(variance(n))

def mode(n):
    counts = Counter(n)
    max_count = max(counts.values())
    return [k for k,v in counts.items()
            if v == max_count]
```

Figure 2: Code generation models produce tokens, not ASTs, and may produce output beyond that requested. This is truncated output from InCoder given just the first highlighted line as the prompt.

We translate two code generation benchmarks that include unit tests for every problem. Figure 1c shows 3 of the 12 unit tests that accompany the problem from Figure 1a. Note that these unit tests are simple assertions: each test asserts that the output value produced by the function matches an expected value.

We judge a generated function correct if it passes *all tests*. Figure 1b shows just one of the solutions generated by Codex for the example prompt. This solution is incorrect because it fails some of the unit tests (Figure 1c). Because the output of the code generation model is stochastic, it is common to sample multiple completions per problem and report an estimated pass rate (§4.2).

## 3 The MultiPL-E Approach

This section describes how we select and prepare languages and benchmarks for MultiPL-E.

### 3.1 Benchmark Selection

There are a number of existing single-language NL2Code benchmarks [9, 10, 11]. We choose to translate HumanEval [1] and MBPP [2] as two of the most widely-used benchmarks.

HumanEval is a good choice of benchmark for several reasons. It is a diverse collection of 164 problems, where all problems have tests to check correctness, and most have examples or doctests as part of the prompt. All of the problems are functions that receive and return first-order values, which facilitates unit testing and test translation. Many also use Python's optional type annotations. Moreover, it is a challenging benchmark: the best model evaluated by Fried et al. [4] achieves only a 36% pass rate on Python.

MBPP is another large, commonly used benchmark of Python problems. As originally formulated, it is a little unusual. Each problem has a description and a list of assertions. The prompt for code generation includes both the description and the assertions, and the generated code

```
# Write a function to fnd the smallest missing
# element in a sorted array. Your code should
# satisfy these tests:
assert smallest_missing([0, 1, 2, 3, 4, 5, 6],
                        0, 6) == 7
assert smallest_missing([0, 1, 2, 6, 9, 11, 15],
                        0, 6) == 3
assert smallest_missing([1, 2, 3, 4, 6, 9, 11, 15],
                        0, 7) == 0
```

(a) An original MBPP prompt: the same assertions are used to test the generated code. (Typo in comment is from the original benchmark.)

```
def smallest_missing(l):
    """
    Write a function to fnd the smallest missing
    element in a sorted array.
    """
```

(b) We add the function signature and hide the test cases to do a more rigorous evaluation.

Figure 3: An original MBPP prompt and how we modify it to standardize evaluation.

is then tested with the same set of assertions. We argue that the HumanEval approach, where test cases are hidden, is a significantly better way to evaluate code generation. We therefore remove the assertions from the MBPP prompts so that we can use them as hidden unit tests. However, with only a problem description, a code generation model is free to make up the name of a function (or not even produce a function). Therefore, we mechanically augment every prompt with a function signature, based on the name and arity implied by the assertions. Figure 3 shows an example of an original MBPP prompt and our modification.

### 3.2 Programming Language Selection

MultiPL-E supports 19 programming languages, which we categorize into four frequency classes (NICHE, LOW, MEDIUM, or HIGH) based on a weighting of TIOBE rank and GitHub frequency (Table 1). Eight of the languages in MultiPL-E had never been used before to measure NL2Code performance; this set includes newer languages (Julia and Swift), older scripting languages (Bash and Perl), and languages for specific applications (Lua and R). Half of the languages are statically type-checked. The broad range of languages in MultiPL-E shows the generality of our compilation approach and allows us to explore how language frequency and language features affect performance (§6). Finally, we ensured that we only selected languages for which the authors had enough expertise to confidently build a compiler and validate its results.

A key feature of MultiPL-E is that it is easy to extend with new models, benchmarks, and languages. To support new languages and benchmarks without manual (and error-prone) effort, we build 18 compilers to translate NL2Code benchmarks written in Python. Writing one of these compilers is straightforward when the target language is similar to Python, but requires care for statically typed languages and even some dynamically typed languages, notably Perl, Bash, and R. §3.4 discusses unsupported languages.

| PL | Static? | GitHub % | TIOBE | Category | LOC |
|---|---|---|---|---|---|
| Bash | × | - | 43 | NICHE | 120 |
| C++ | ✓ | 7.0 | 4 | HIGH | 244 |
| C# | ✓ | 3.1 | 5 | MEDIUM | 149 |
| D | ✓ | - | 35 | NICHE | 117 |
| Go | ✓ | 7.9 | 12 | MEDIUM | 210 |
| Java | ✓ | 13.1 | 3 | HIGH | 153 |
| JavaScript | × | 14.3 | 7 | HIGH | 45 |
| Julia | × | 0.1 | 28 | NICHE | 125 |
| Lua | × | 0.2 | 25 | NICHE | 43 |
| Perl | × | 0.3 | 17 | LOW | 49 |
| PHP | × | 5.3 | 11 | MEDIUM | 50 |
| R | × | 0.05 | 19 | LOW | 98 |
| Racket | × | - | - | NICHE | 38 |
| Ruby | × | 6.2 | 15 | MEDIUM | 41 |
| Rust | ✓ | 1.1 | 22 | LOW | 147 |
| Scala | ✓ | 1.7 | 32 | LOW | 152 |
| Swift | ✓ | 0.7 | 10 | LOW | 479 |
| TypeScript | ✓ | 9.1 | 33 | HIGH | 117 |

Table 1: MultiPL-E languages by frequency, as calculated by GitHut 2.0 and the 2022 TIOBE Programming Community index; the LOC column indicates the number of semantic lines of code in our compiler.

### 3.3 Compiling Python Benchmarks

A MultiPL-E compiler is significantly easier to build than a complete compiler. To translate a benchmark problem, we only need to compile function signatures and unit tests (not arbitrary statements and expressions). Our compilers preserve comments, since they contain the natural language description for the NL2Code task; however, we automatically rephrase them to replace Python-specific terminology.

#### 3.3.1 Compiling Unit Tests

MultiPL-E supports any unit test where the input and output to the test are *first-order values*. In Python, these include constants and data structures such as lists, tuples, and dictionaries, but exclude values such as `lambda` expressions.[6] HumanEval and MBPP unit tests apply the model-generated function to a first-order value, and compare the result with an expected first-order value.

Each MultiPL-E compiler has a recursive function that compiles Python values to the target language's values. Even for a dynamically typed target, this value-to-value compilation requires care, because not all Python value types have perfect analogues in every target. For example, we compile both tuples and lists to JavaScript arrays, since JavaScript lacks a canonical tuple type. We also support dynamically typed targets where the compilation strategy is less obvious. For example, when the target is R, it may appear natural to compile Python lists to R lists: both kinds of lists can be nested and allow heterogenous values. However, R's vector type is much more commonly used (data frames are made of vectors). Unfortunately, vectors must be homogeneous and cannot be nested, so not all Python lists can be translated to vectors. For example, an argument typed `List[Int]` can be translated to a vector, but a nested list cannot. In order to more closely match the token distribution of idiomatic R code seen during training, our R compiler uses types (described below) to identify

6. We do not support testing higher-order functions, but support generated code that uses higher-order functions.

(a) Original Python assertion.

```
assert lsi([0]) == (None, None)
```

(b) Equivalent R.

```
if(!identical(lsi(c(0)), c(NULL, NULL))){
    quit('no', 1)}
```

(c) Equivalent JavaScript.

```
assert.deepEqual(lsi([0]), [void 0, void 0]);
```

Figure 4: Example of a translated assertion.

homogenous list values and maps them to vectors using `c()`—even though R is dynamically typed.

The final step of compiling tests is to choose an appropriate test for equality. The meaning of equality operators varies across programming languages. Python's `==` operator checks *deep equality*, i.e., item-by-item equality within data structures. Deep equality is the appropriate choice for unit tests. In some languages, we need to import equality-testing functions from testing libraries, as in the JavaScript example shown in Figure 4.

#### 3.3.2 Translating Types and Type Inference

Compiling a function signature to a dynamically typed language is straightforward, but requires care when the target is typed. Most typed languages require argument and return type annotations. Python has optional type annotations. Thus if a benchmark has type annotations, we can translate them to types in a target language. Fortunately, a large subset of the HumanEval benchmarks employ Python's optional type annotations. We introduce type annotations to the few that do not. None of the MBPP benchmarks have type annotations. Instead of manually adding annotations to 400+ benchmarks, we infer the types of the values that appear in the MBPP assertions.

Translating types and typed values is subtly different for every language. For example, five HumanEval problems use types such as `Any` which cannot be translated to most traditional statically typed languages (e.g., C++ and Rust). We fail to compile these few problems to these languages.

Another problem arises when compiling to languages with algebraic datatypes or discriminated unions. For example, consider translating the Python type `Optional[Int]` to Rust, Swift, or Scala. The analogous type in the target language is an algebraic datatype. This means that when the Python number `n` has type `Optional[Int]` it must translate to the value `Some(n)`. Optional values are very common in Python benchmarks, and we use this approach extensively.

Finally, many typed languages require type annotations in data structures, which appear in unit tests. For example, C++ vectors require an annotation specifying their element type, and numbers in Rust (sometimes) require a type suffix. We perform limited local type inference to calculate these types from the type of the function signature to ensure that the unit tests always compile successfully.

#### 3.3.3 Translating Doctests

Python *doctests* are a standard format for examples in documentation. While many of the HumanEval prompts include

(a) Original Python docstring from HumanEval #95.

```
Given a dictionary, return True if all keys are
strings in lower case or all keys are strings in
upper case, else return False. The function should
return False is the given dictionary is empty.
```

(b) Terminology translated to Perl.

```
Given a hash, return 1 if all keys are strings
in lower case or all keys are strings in upper case,
else return "". The function should return "" is
the given hash is empty.
```

Figure 5: A Python docstring and its Perl translation. Errors (e.g., "is" for "if") are from the original benchmark.

examples, not all of them are validly formatted doctests. We standardize examples to the Python doctest format (">>>" prepended). We apply value-to-value compilation to the doctests as we do for unit tests. However, since not all languages have an equivalent doctest format, we keep the Python format for all target languages.

#### 3.3.4 Translating Python Terminology in Prompts

Different programming languages use different terminology to refer to the same concept. For example, a Python *list* is closest to a JavaScript *array* or a Rust *vector*. To mitigate the impact of these differences, we identify Python-specific terminology in the natural language portion of the prompt, and translate it to the most natural equivalent for the target language. Figure 5 shows an example of a prompt translated from Python to Perl. Notably, Perl not only lacks Booleans, but uses 1 for true and the empty string for false.

We conservatively avoid translating number types. Although some target languages use different terms for floats and integers, the term *integer* is commonly used in a mathematical sense rather than in reference to the Python type.

### 3.4 Limitations of Our Approach

A handful of benchmarks cannot be easily translated using the MultiPL-E approach. Of the 164 original HumanEval benchmarks: (1) we exclude 3 benchmarks that have Python helper functions in their prompt; (2) we modify 2 benchmarks to use unit tests instead of randomized testing; and (3) for certain typed languages, we fail to compile up to 5 benchmarks with untranslatable types. These changes do not lead to significantly different results for Python (§5.2.1).

Our approach can be generalized to additional programming languages, so long as the target language has natural analogues for the Python data types used in the benchmarks. We do not include two previously studied languages, C [7] and SQL [28, 29] because they do not meet this criterion. SQL queries and tables define relations and not functions, which our benchmarks define. The problem with pure C is that there are no canonical types for dictionaries and lists. This is unlike C++, which has STL data structures, and our C++ prompts include the STL headers. A prompt for C would have to include one of several non-canonical libraries and would be very sensitive to the choice made.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3267446



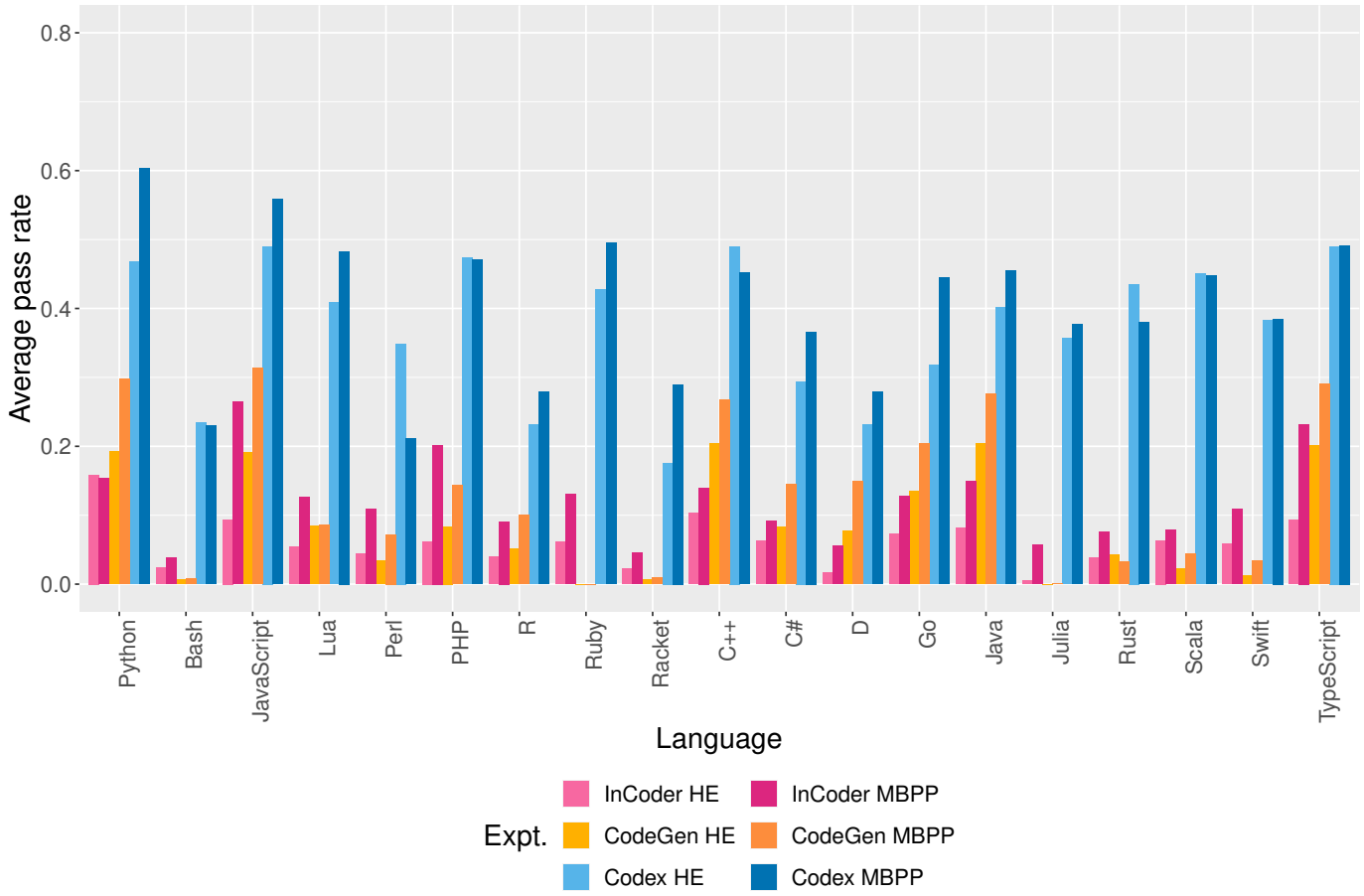


Figure 6: Pass@1 rates for all languages in MultiPL-HumanEval and MultiPL-MBPP. From left to right: InCoder, CodeGen, Codex.

### 3.5 Validating Prompts and Test Suites

Since we automatically generate thousands of prompts and test suites, we validated the quality of generated prompts and test suites in several ways.

First, we identified a subset of 23 problems on which we manually validated the generated prompt and test cases for every language. We selected these problems because they exercised a variety of syntactic and semantic features. For example, we selected problems that used different types and compositions of types, including lists, dictionaries, tuples, unions, the any type, the optional type, and nested types. We also selected problems with test cases that used particular kinds of values, including empty lists, the `None` value, and literal newline characters in strings. In addition, we performed language specific validation when necessary, e.g., with Bash.

Second, on the aforementioned 23 problems, we manually checked several sample completions produced by Codex. This helped us ensure that the solutions were in the right programming language, and develop the set of stop tokens for each language.

Finally, we validated our evaluators writing test cases for every language, including tests that we expected to fail. Moreover, since we classify failures as syntax errors or runtime errors, we constructed tests that failed in both ways.

## 4 Code Generation Models

We evaluate three state-of-the-art code generation models, each of which use a Transformer architecture [30] and are trained with a language modeling objective on a mixture of natural language and code. We evaluate the largest, best-performing versions of each of these three models.

### 4.1 Models

**InCoder** InCoder [4] is a 6.7B parameter language model trained using a causal masking objective [31]. It supports both code infilling and code completion; we test only the latter. InCoder was trained on 159 GB of deduplicated, filtered code from Github (around a third in Python) and 57 GB from StackOverflow.

**CodeGen** CodeGen is a 16.1B parameter language model trained with a next-token prediction objective. We evaluate the multilingual CodeGen model, which was trained first on The Pile [32], a 825 GB dataset of mostly natural language text with around 8% Github-scraped code. The model was further trained (fine-tuned) on a 6 programming language subset (C, C++, Go, Java, JavaScript, and Python) of the BigQuery code dataset.[7]

7. https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code

**Codex** Codex is a GPT-3 language model fine-tuned on code. [1] describe a 12B parameter version of Codex fine-tuned on 159 GB of deduplicated, filtered Python code from Github. We use the more recent 175B parameter `code-davinci-002` model, which is trained on multiple languages. Details of its training set are not publicly known. We use the OpenAI API to query Codex.

### 4.2 Metrics

For each language, we calculate pass@$k$ using the methodology employed by [1] and subsequent work. Intuitively, pass@1 is the likelihood of the model producing a completion that passes *all unit tests*, pass@10 is the likelihood of any one of 10 completions passing all unit tests, and so on. We calculate pass@1 with temperature $0.2$, and use temperature $0.8$ for pass@10 and pass@100. For statistical reliability, we take 200 completions at each temperature and calculate average pass rate using the unbiased sampling estimator presented in [1].[8]

## 5 Evaluation

In this section, we present the results of evaluating Codex, InCoder, and CodeGen on MultiPL-HumanEval and MultiPL-MBPP. We fit mixed-effects models to evaluate the statistical significance of the differences between groups that we report below [33]. Appendix C has a full description of each statistical model with its estimate table.

### 5.1 Effect of Model Size and Training Data

We expect to find that Codex will outperform CodeGen, and CodeGen will outperform Incoder. First, the most important factor is model size: Codex has 175B parameters, CodeGen 16.B parameters, and InCoder 6.7B. A larger model has higher capacity and can be trained effectively on more data. Second, the training time and data also matters. The CodeGen corpus is nearly 2TB, the InCoder corpus 159GB, and the Codex corpus is unknown. Apart from size, the training data for InCoder and CodeGen have different compositions of programming languages. CodeGen is trained on six languages, InCoder is trained on more, but most of these extra languages have negligible quantities of data. Finally, we expect InCoder's performance on left-to-right generation to suffer slightly because a portion of its training is on fill-in-the-middle problems. Although InCoder did not perform an ablation, subsequent work by some of the authors of InCoder and this article show that training to fill-in-the-middle has a small impact (1%) on pass@100 rates [34].

### 5.2 MultiPL-HumanEval results

We explore the code generation abilities of the three models on our translated version of HumanEval, MultiPL-HumanEval. Figure 6 shows the by-language performance of each model on both benchmarks. Note that the Codex model is over 10x and 20x larger than CodeGen and InCoder respectively, so it performs much better as expected.

We find reliable differences between Codex pass@1 rates on MultiPL-HumanEval for Python and all but 4 languages: C++, JavaScript, Scala, and TypeScript. For these languages, Codex performance is similar to Python.

CodeGen performs best on the languages included in its fine-tuning dataset (Python, JavaScript, Java, C++, and Go). It also performs well on TypeScript, likely due to its similarity to JavaScript. A mixed-effects model finds reliable differences in pass@1 rates on MultiPL-HumanEval between all languages and Python, except Ruby, where performance is so low that the model fails to find a reliable estimate.

InCoder performs significantly better on the Python version of MultiPL-HumanEval than all of the other language versions ($p < 0.001$ for all languages).

#### 5.2.1 Python Results and Replication

Our InCoder results on Python exactly replicate its previously reported performance on HumanEval [4]. We measure a slightly higher pass@1 rate for CodeGen than what is reported in [3] (19.2% compared to 18.3%).[9] These findings show that the small standardization changes we made to the HumanEval benchmarks do not significantly affect model performance.

We evaluate a more recent Codex model (`code-davinci-002`) than the original paper and observe a large improvement on Python: a pass@1 rate of 45.9%, compared to 28.8% reported earlier [1]. Our pass@1 rate on the Python HumanEval subset replicates what is reported for `code-davinci-002` in [35].[10]

#### 5.2.2 Codex Performs Best on JavaScript

Codex's performance on JavaScript is better than its performance on Python, though the difference is not significant (+2.3%; $p = 0.43$). Codex also achieves a pass@1 rate higher than 40% on C++, Java, TypeScript, PHP, Ruby, Rust, Scala, and Lua. Since its training set is not public, we cannot independently verify that the model was not trained on Python-HumanEval solutions. However, that Codex matches or exceeds its Python performance on 18 new languages suggests a negligible impact of any train/test overlap.

CodeGen also performs well on JavaScript and TypeScript, though the latter is not included in its fine-tuning dataset. InCoder's performance is the weakest. Like the other models, it performs better on more frequently-used languages (Python) than less popular ones. However, unlike Codex and CodeGen, it does not match its Python performance on any other language.

#### 5.2.3 Performance by Language Frequency

Figure 7 shows MultiPL-HumanEval pass@1 rates for each model, grouping languages by frequency. All three models perform best on high frequency languages.

Although we find reliable differences in Codex pass@1 rates between the MEDIUM, LOW, and NICHE languages

8. We note that pass@1 rates appear to stabilize around 20 samples, suggesting that future work could achieve a stable estimate with a less computationally costly sample size.

9. We note that [3] calculates the pass@1 rate for 3 temperatures, and reports the best result without specifying the temperature. Consequently, it's not clear whether the 18.3% pass@1 rate they report is measured at the 0.2 temperature that we use.

10. We note that [35] also reports results for InCoder and a monolingual version of CodeGen, but with sampling differences that make the pass@1 rates difficult to compare.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3267446



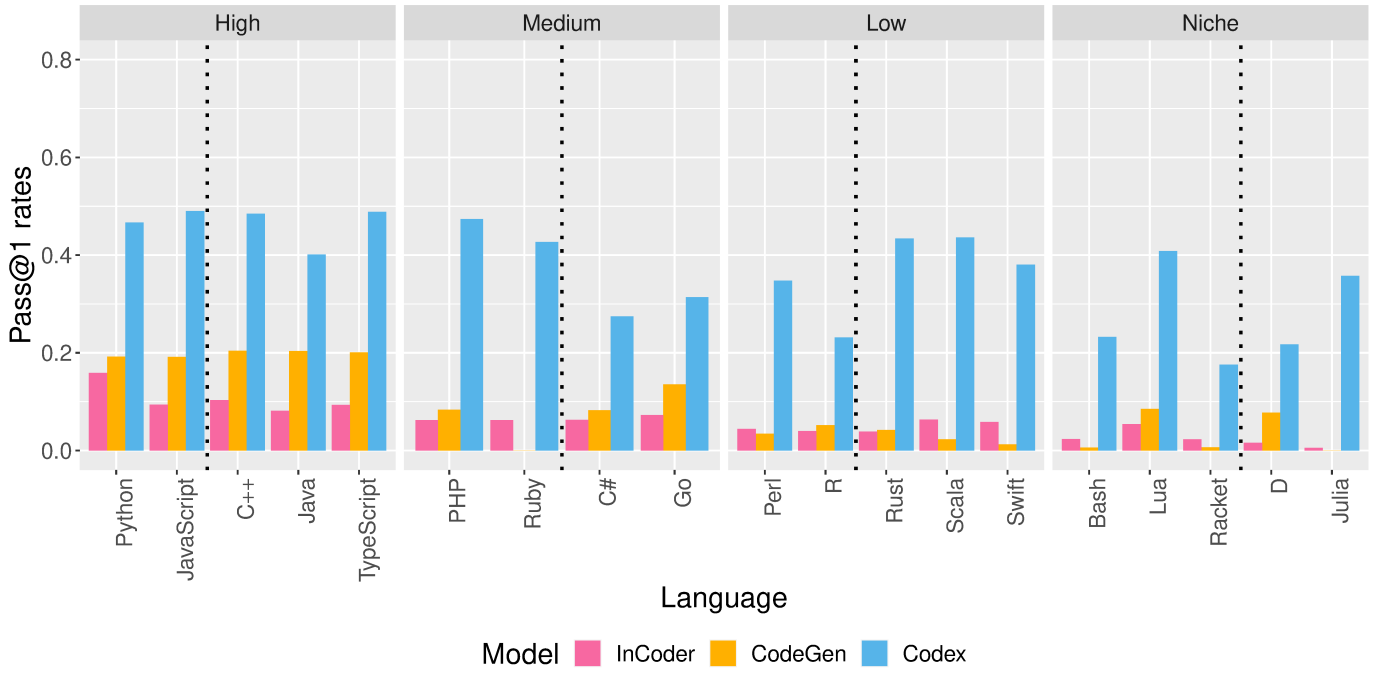


Figure 7: Model performance on MultiPL-HumanEval by language frequency and type-checking. Languages to the left of dashed line are dynamically typed, and languages to the right are statically typed. Julia is unusual: type annotations are the norm and methods are type-checked at runtime.

when compared to the HIGH category ($p = 0.006$; $p < 0.001$; $p = 0.002$), we observe that Codex performs very well on some LOW and NICHE languages. For instance, Lua is the 9th-best language in our Codex evaluation, although it only appears in 0.2% of GitHub activity and is not in the TIOBE Top-20. CodeGen also performs well on Scala, Rust, and Julia.

Our evaluation therefore shows that contemporary code generation models may be useful even for developers working with less commonly used programming languages.

#### 5.2.4 *Perplexity and Code Correctness Do Not Correlate*

Xu et al. [7] report Codex perplexity scores for 11 of our 18 languages. We do not observe a strong correlation between Codex pass@1 scores and their perplexity scores (Figure 8). Notably, perplexity is highest for JavaScript and TypeScript, while we find that Codex performs *best* on these languages. Therefore, perplexity may not be a reliable evaluation metric for NL2Code. One caveat is that [7] likely evaluate an older Codex model, since they report substantially lower pass rates for Python.

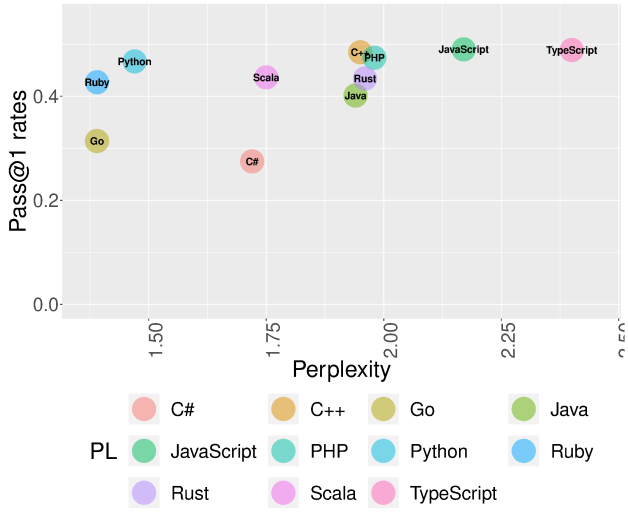


Figure 8: Codex HumanEval pass@1 rates versus perplexity scores reported in [7].

### 5.3 MultiPL-MBPP results

Figure 6 shows the by-language performance for Codex, CodeGen, and InCoder on our translated version of the MBPP benchmark, MultiPL-MBPP. Codex performs strongest on the Python problems, but, as with MultiPL-HumanEval, does well on several other languages, including JavaScript. A mixed-effects model finds significant differences in Codex pass@1 rates between Python and all other languages in MultiPL-MBPP.

As with MultiPL-HumanEval, CodeGen performs best on the MultiPL-MBPP languages included in its fine-tuning set: Python, JavaScript, C++, Java, and Go. It performs almost as well on TypeScript as on JavaScript. A mixed-effects model finds significant differences in CodeGen pass@1 rates between Python and all languages except Ruby, where performance is so low that the model fails to find a reliable estimate.

Unlike with MultiPL-HumanEval, on MultiPL-MBPP, InCoder's performance on TypeScript, JavaScript, and PHP actually exceeds its performance on Python. InCoder's Python pass@1 rate is similar on MultiPL-HumanEval and MultiPL-MBPP, one of the few instances where MBPP per-

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3267446



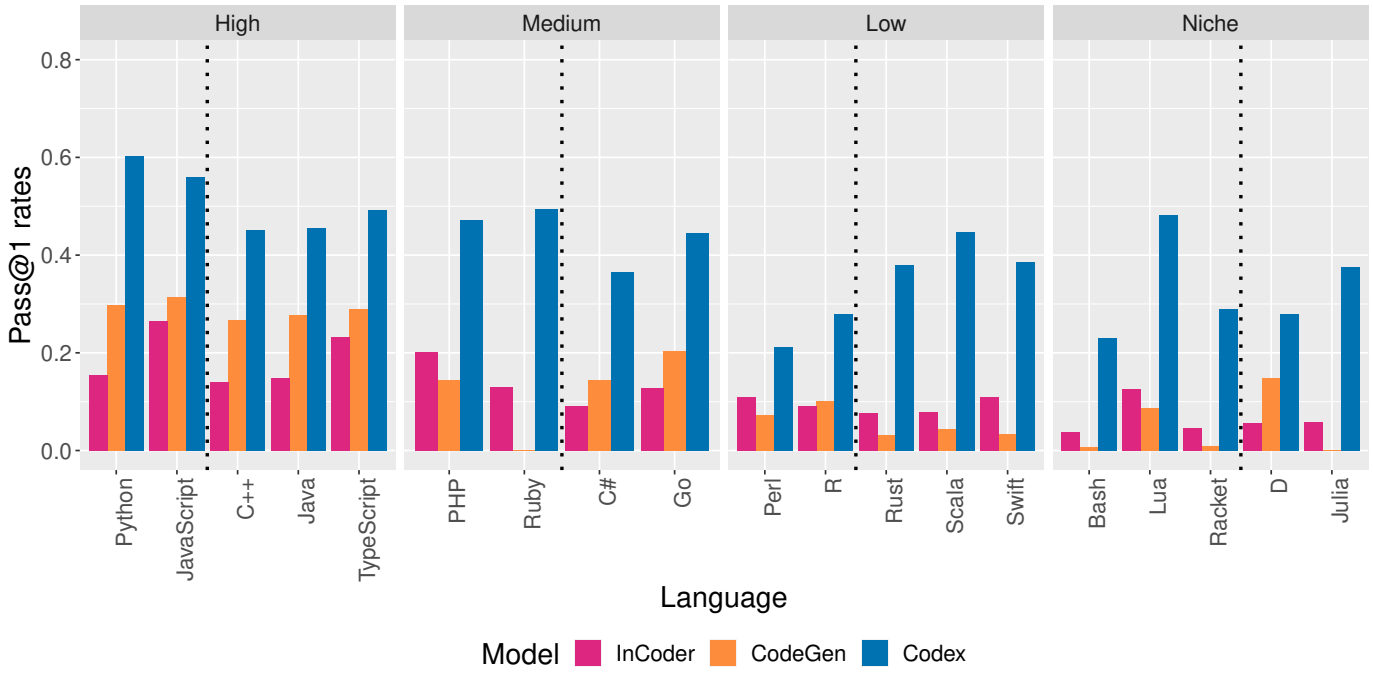


Figure 9: Model performance on MultiPL-MBPP by language frequency and type-checking. Languages to the left of dashed line are dynamically typed; languages to the right are statically typed.

formance is not considerably better than HumanEval. A mixed-effects model finds significant differences in InCoder pass@1 rates for all languages, with positive coefficients for TypeScript, JavaScript, and PHP.

Note that we do not have any doctests in MBPP (§3.1). The performance differences between MultiPL-MBPP and MultiPL-HumanEval on certain languages may relate to this, as we discuss in more detail in §6.1.

#### 5.3.1 MBPP is Less Challenging Than HumanEval

MBPP appears to be a less challenging benchmark than HumanEval. The MultiPL-MBPP pass@1 rate is higher than the MultiPL-HumanEval pass@1 rate for all but 6 of our 57 model/language pairs. This is despite the fact that MBPP does not provide doctests in any prompts, which, as we show in §6.1, affects performance for some languages.

This suggests that HumanEval may be a more useful benchmark suite than MBPP for the community, as it provides an equally good or better indication of model performance with a more computationally efficient sample size.

#### 5.3.2 Python Results and Replication

Our Python MBPP pass@1 rates for Codex are slightly higher than what is reported in [35] (60.3% compared to 58.1%). [35] prompts with a function signature and docstring, even though the original MBPP problems do not include function signatures; we also include function signatures, which we infer from the provided test cases.

Our Python MBPP results for InCoder are lower than what is reported in the original paper (15.5% compared to 19.4%) [4]. We calculated pass@1 rates for MBPP differently than Fried et al. [4] in two key ways. First, since the original MBPP benchmarks do not include function signatures, Fried et al. [4] prompts InCoder with the MBPP docstring only. We infer function signatures for MBPP problems from the provided test cases, as described in §3. Second, Fried et al. [4] reports computing pass@1 rates for MBPP using a single completion, rather than computing the unbiased sampling estimator with 200 samples as described in Chen et al. [1], as we do. We suspect this leads to inflated pass@1 rates.[11]

#### 5.3.3 Performance by Language Frequency

Figure 9 shows model performance on MultiPL-MBPP by language frequency. As with the MultiPL-HumanEval benchmark, models generally perform better on more common languages. However, Codex performance on MultiPL-MBPP is robust on many MEDIUM, LOW, and NICHE languages, such as Lua and Scala. CodeGen performs surprisingly well on the D version of MBPP, a niche language not included in its fine-tuning dataset.

We find reliable differences in Codex pass@1 rates between languages in the MEDIUM, LOW, and NICHE categories when compared to the HIGH category ($p = 0.007$; $p < 0.001$; $p < 0.001$).

#### 5.3.4 Comparing Model Performance by Problem

Although Codex outperforms CodeGen and InCoder, one may wonder if the latter models can solve problems that Codex cannot. This type of comparison is subtle since all of these models are non-deterministic, and the pass@$k$ metric estimates the likelihood of a prompt producing a working program. We can consider the subset of prompts

11. With a single sample, the pass@1 rate for an individual problem will be 0% or 100%. Assuming that the 200 samples are fairly homogeneous but not identical, as we observe is usually the case, the unbiased sampling pass@1 rate for an individual problem is rarely 100%, since this would require all 200 samples to be correct.

where Codex never produces a working program. There are exactly 3,000 out of over 10,000 such prompts, across all languages and benchmark suites. For these prompts, we calculate the 95% confidence interval for pass@1 with CodeGen and InCoder and find that 3% of them have pass@1 $> 0$. Moreover, the mean value of pass@1 for the aforementioned 3% of prompts is $0.16$ and $0.14$ for CodeGen and InCoder respectively. Thus the likelihood of CodeGen or InCoder solving a problem that Codex cannot is very low.

### 5.4 Summary

On the whole, our results replicate previously reported model performance on code generation for Python. We benchmark three state-of-the-art models on 18 additional languages, most of which have never been evaluated before. Surprisingly, we find remarkably good model performance on some lower-frequency languages, such as Lua. We also find that performance on JavaScript and TypeScript is consistently high and occasionally exceeds Python, even though the benchmarks we explore originated in Python.

### 5.5 Cost

We use CodeGen and InCoder "off-the-shelf" from Hugging Face Transformers and estimate that we spent about 2 years of time on a variety of Volta/Ampere GPUs. We used the Codex model when it was in limited beta, and it is now available from Azure at 10 cents / 1,000 tokens. The total length of all Codex completions in our dataset is nearly 1.5 billion characters. With the OpenAI estimate of 0.25 characters per token, our experiment would have cost approximately $37,000 if we had had to pay for it.

## 6 Factors in Code Generation Success

In this section, we explore factors that impact code generation success. Focusing specifically on the MultiPL-HumanEval benchmark suite, we report results from a number of follow-up experiments, including an ablation study of MultiPL-E's translation components and finer-grained examinations of language features and prompt translation choices. We also provide a fine-grained analysis of the kinds of errors that arise in NL2Code across several languages.

### 6.1 Ablation Study

Our compilers target multiple distinct regions of the prompt for each problem. We explore the impact of each component in an ablation study of our MultiPL-HumanEval benchmark suite with Codex. We ran four versions of the MultiPL-HumanEval prompts, with some or all regions translated:

- **Original Prompt**: does not translate doctests or natural language terminology (e.g. prompts as in HumanEval);
- **Test-only Translation**: translates doctests but not Python-specific terminology;
- **Full Translation**: translates unit tests, doctests, and Python-terminology in the prompt; and
- **No Doctests**: removes doctests and does not translate natural language terminology.

| Language | Implicit | Explicit |
|---|---|---|
| D | 0.22 | 0.17 |
| R | 0.23 | 0.21 |
| Racket | 0.18 | 0.20 |

Table 2: pass@1 with Codex on three languages where Codex underperforms. The *Implicit* column is the full translation, and the *Explicit* column adds an explicit cue about the desired language.

For Codex's pass@1 results, translating doctests and Python-specific terminology has little impact on better-performing languages (Figure 10). However, translating these components seems more important for certain languages: Bash, PHP, Perl, R, Rust, Swift, and TypeScript.

We note that six of these languages are ones where Codex does not perform substantially better on MultiPL-MBPP than MultiPL-HumanEval (Figure 6). The performance degradation observed for these languages when doctests are removed from the MultiPL-HumanEval problems suggests that the worse than expected performance on MultiPL-MBPP could be due to the lack of doctests in that benchmark suite.

Overall, we find significant differences between the **Full Translation** and **Test-Only Translation** experiments ($p = 0.03$), and between **No Doctests** and **Test-Only Translation** ($p < 0.001$), but not between **No Translation** and **Test-Only Translation** ($p = 0.2$). This suggests that the Python terminology translation has a small but reliable effect, and that the presence of the doctests is important, though their translation is not.

### 6.2 Explicitly Prompting with Language Name

Our prompts do not explicit specify the name of a programming language and instead relies on the models to infer the desired language from other cues in the prompt (§2.1). We run a small ablation study with Codex using three programming languages on which Codex performs poorly: D, R, and Racket.[12] For each of these languages, we take the original prompt (specifically, the full translation that translates doctests and terminology) and add "Generate in *language*" as a comment before the original prompt. Table 2 shows pass@1 rates with and without this explicit language cue. The results are inconclusive across languages: Codex's performance on Racket improves slightly, but is slightly worse on D and R.

### 6.3 Type Annotations

One may conjecture that type annotations improve model performance by constraining the code generation search space. Or, perhaps, they might hurt performance by complicating the task, since the model must generate correct type annotations.

In Figure 7 and Figure 9, the dashed lines in each category separate languages with type annotations (left) from languages without (right). We observe no overall effect of type annotations on Codex pass@1 rates on MultiPL-HumanEval ($p = 0.33$) or MultiPL-MBPP ($p = 0.23$).

12. We take care to add this after the `#lang racket` line for Racket.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3267446



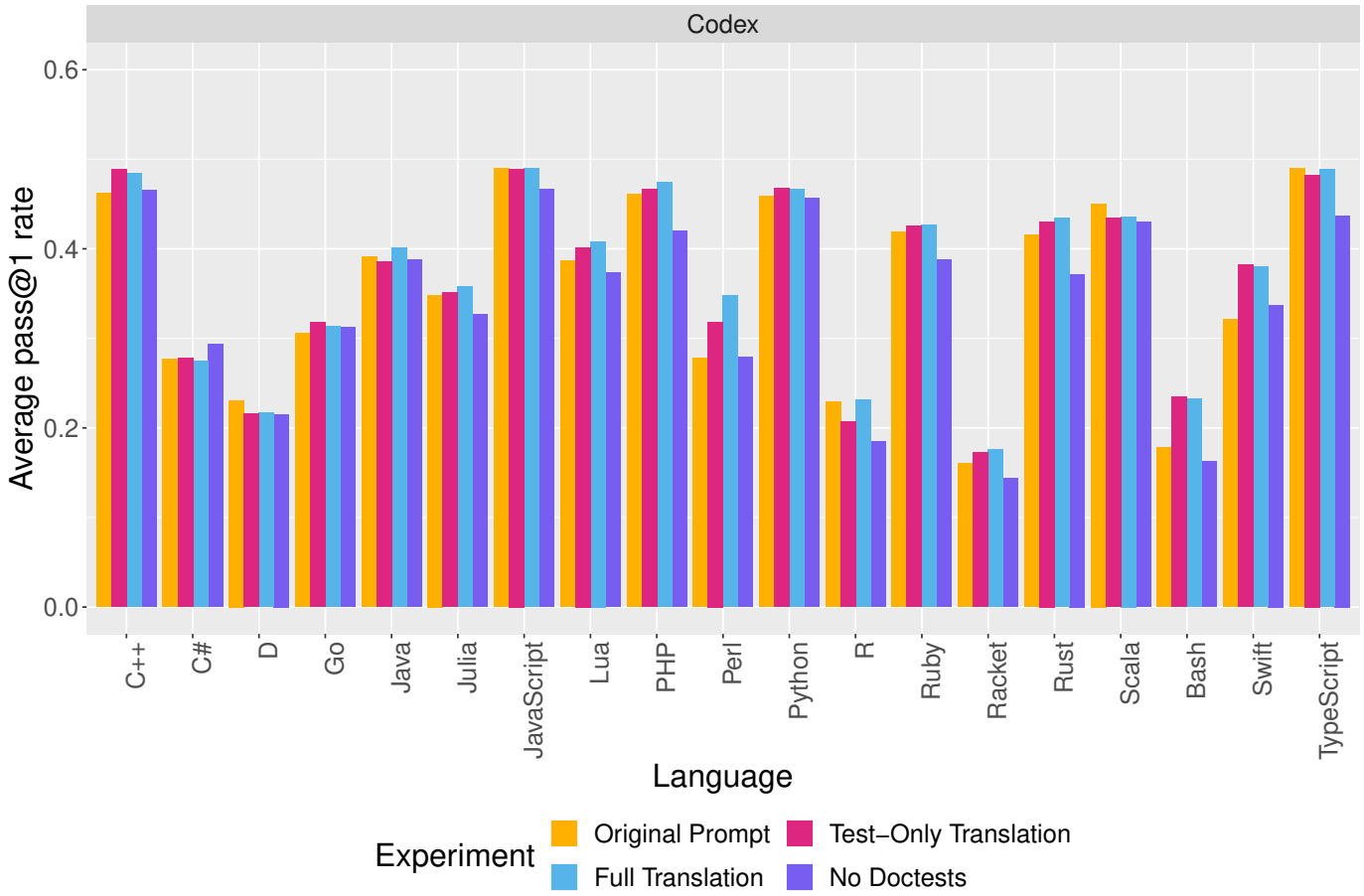


Figure 10: Ablation study of translation components, showing Codex pass@1 with original prompts; translated doctests; translated text and doctests; and doctests removed.

To explore the impact of type annotations at a more fine-grained level, we run a series of follow-up experiments using the MultiPL-HumanEval benchmark suite. We focus on two languages: Python, which allows optional type annotations, and TypeScript, a gradually typed cousin of JavaScript. Gradual typing allows us to weaken type annotations or even configure the TypeScript compiler to ignore all type errors.

#### 6.3.1 *Precise types improve TypeScript performance*

TypeScript has an "Any" type, which is compatible with all types. We run Codex on a variation of the MultiPL-HumanEval TypeScript prompts where all types in the function signature are translated to "Any". We find that the loss of precise types hurts performance on TypeScript (-2.5%; $p < 0.001$).

#### 6.3.2 *TypeScript type errors correlate with runtime errors*

Even gradual type-checking can reject programs that would in fact run without error. We run the Codex-generated TypeScript programs without first checking types. We observe no significant difference in pass@1 rates ($p = 0.14$), suggesting that typed programs are rejected for genuine errors.

#### 6.3.3 *Type annotations* do not *affect Python performance*

We run a similar experiment with Codex and Python, where we remove all the type annotations from the MultiPL-HumanEval prompts. We find that this has no significant effect on Codex's pass@1 rate for Python ($p = 0.23$).

We interpret these results as evidence that type annotations do not guide search in general, since they do not improve Python performance, but that informative types are necessary for languages where type annotations are standard, perhaps in order to fit the token distribution of high-quality typed code seen in training.

### 6.4 Sensitivity to Compilation Choices

Each MultiPL-E compiler makes small choices about how to translate prompts that could have an impact on performance. We explore some of these choices below.

#### 6.4.1 *Comment style affects performance*

Most programming languages have several comment styles (e.g., single-line vs. multi-line). To investigate their impact, we ran follow-up experiments with Codex on the MultiPL-HumanEval benchmark suite for two languages: PHP (MEDIUM) and Racket (NICHE).

Our original prompts use single-line comments for both PHP and Racket, following conventional style. We re-ran Codex on versions of the MultiPL-HumanEval problems for both languages using multi-line comments instead. This improves the pass@1 rate for Racket (+1.9%, $p < 0.001$), but decreases it for PHP (-3.1% , $p = 0.001$).

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3267446



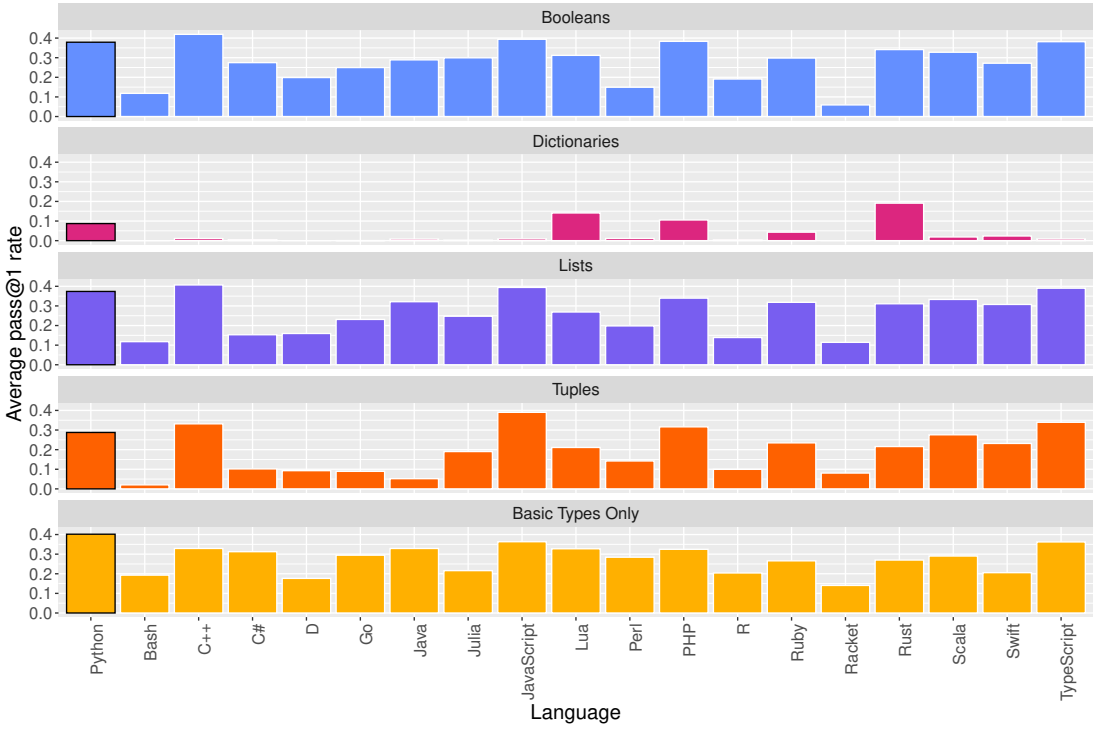


Figure 11: Impact of programming language features on Codex pass@1 performance by language

#### 6.4.2 *Naming arguments improves performance for Perl*

Functions in Perl do not have formal named arguments. Instead, all arguments are passed in a special array. Our compiler to Perl produces a prompt that pops elements off the special array and names them, with the expectation that this would improve model performance.

We ran a follow-up experiment on a version of the MultiPL-HumanEval problems for Perl where we omit this argument-naming prompt, so the model has to infer everything about arguments from the natural language description and examples. This significantly lowers Codex's pass@1 rate (-8%; $p < 0.001$).

In summary, our results show that code generation performance can be sensitive to prompt engineering choices for both high and low frequency languages.

### 6.5 Impact of Language Features

One challenge of extending existing benchmarks to new programming languages is that not all programming languages have the same features. Although the MBPP and HumanEval benchmarks consist of relatively simple functions, they exercise a variety of datatypes, not all of which have a straightforward equivalent in all programming languages in our dataset.

To explore whether model success is impacted by the Python language features used in the program, we categorized all problems from the HumanEval benchmark suite into groups based on the Python language features used in their type annotations: Booleans, dictionaries, lists, tuples, or none of the above.

Figure 11 shows the performance by language on each type of problem. A mixed-effects model finds no significant effect of problem type, when programming language is treated as a random effect.

Many models appear to struggle with questions involving tuples. Some of these are on languages that lack a native tuple type, such as Java. However, JavaScript performs well despite lacking tuples.

Although many languages show poor performance on dictionary problems, there are only 3 problems in this category, so these results should be interpreted with caution.

### 6.6 Fine-grained Error Analysis

Code generation models generate many more failing programs—programs that produce errors or fail to pass unit tests—than programs which run successfully. This section presents a detailed evaluation of errors present in the Codex-generated completions for MultiPL-HumanEval problems in 4 languages: Python, C#, Swift, and Racket. See Appendix D for a full categorization.

We first identified specific error labels for each language and then grouped them into themes (e.g. "NullReference"). We produced five general error categories: RUNTIME, STATIC, TYPE, LANGUAGE, and MODEL. We group similar error sources together across languages, even if they occur in different contexts: for example, calling a function with a value of the wrong type may fail at compile-time or run-time, depending on the language's type system.

The most common STATIC theme across all languages is "UndefinedIdentifier", which contains errors related to referencing non-existent terms (e.g., Figure 12a). These errors can be caused in many ways – calls to functions not in the local context, use of Python-like keywords, or calls to methods from external libraries that were not imported.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3267446



```
public static string Intersection(
  Tuple<long, long> interval1,
  Tuple<long, long> interval2) {
  long start = Math.Max(interval1.Item1, interval2.Item1);
  long end = Math.Min(interval1.Item2, interval2.Item2);
    if (start > end) {
      return "NO";
    }
  long length = end - start + 1;
  return IsPrime(length) ? "YES" : "NO";
}
```

(a) A C# completion refers to a non-existent `IsPrime` method (HumanEval #127).

```
def find_closest_elements(numbers):
  # YOUR CODE HERE
  raise NotImplementedError()
```

(b) A Python completion that always fails (HumanEval #20).

```
(define (solve s)
(if (string-contains s #\a)
    (string-map (lambda (c) (if (char-alphabetic? c)
                                (char-upcase c)
                                (char-downcase c)))
                s)
    (string-reverse s)))
```

```
### [Reverse the order of words in a string](...

```racket
#lang racket
```

(c) A Racket completion that turns into Markdown (HumanEval #161).

```
func starts_one_ends(n: Int) -> Int {
  if n == 1 {
      return 2
  }
  if n == 2 {
      return 4
  }
  return starts_one_ends(n - 1) +
         starts_one_ends(n - 2)
}
```

(d) A Swift completion that does omits required argument labels in function (HumanEval #83).

Figure 12: Some examples of different types of errors produced by Codex. For brevity, we have removed the problem descriptions (comments) from the prompts.

Some errors in the RUNTIME category mimic those we expect from software engineers (e.g., index-out-of-range errors). However, others are unlike human mistakes. Notable themes in the latter group (MODEL) include generating code that throws exceptions on purpose (e.g., Figure 12b) and generating code in an entirely different language. For instance, Codex frequently generates Markdown code for Racket problems. Although we don't have access to the Codex dataset, we suspect that Racket is not well-represented in the dataset. We posit that these errors occur because Racket files begin with a language declaration (`#lang racket`) that is easily mistaken for a Markdown heading (e.g., Figure 12c).

Finally, the category LANGUAGE includes multiple themes related to the specifics of the target language itself. The "LanguageSpecific" theme contains idiosyncratic errors such as the requirement of labeled arguments in Swift (e.g., Figure 12d). An additional theme ("DoesNotKnowSyntax") includes errors in Racket caused by incorrectly generated core language constructs.

## 7 THREATS TO VALIDITY

Our work translates two Python code generation benchmarks into 18 other languages and evaluates the performance of three code generation models on the translated benchmarks.

The principal threat to validity is that the (translated) benchmarks may not be representative of the kinds of problems that programmers typically solve in each languages. For example, we evaluate both scripting languages (e.g., Python and JavaScript) and systems languages (e.g., C++ and Rust) on the same task, but programmers frequently use these languages for very different tasks. We characterize the HumanEval and MBPP benchmarks as a mix of basic programming problems and straightforward interview questions. Thus, performance on benchmarks may not accurately represent real-world performance.

Code generation models are sensitive to small changes in how prompts are designed, as we show in our exploration of prompt design choices for three of our languages (§6.4). It is likely that the pass rate on individual languages could be improved with even more language-specific effort. We do provide an ablation study on prompts for all languages in our dataset (§6.1) to investigate the impact of our different translation components. All our prompts use >>> to mark examples, which is Python doctest notation. It is possible that a different marker may improve performance on some other languages.

The quality of generated code is also sensitive to decisions about how to sample completions (§2.2). We use the same sampling configuration that is used in most prior work on code generation. Empirical results show these settings are optimal for Python [1], but it is possible that different settings would be better for other programming languages.[13] However, in a practical deployment of a multi-language code generation model, it may not be feasible to adjust the sampler for every input language.

## 8 RELATED WORK

In this section we focus on related work on evaluating neural code generation models.

13. This would be a very resource-intensive experiment, beyond the scope of an academic group. The original experiment on sampler configurations by Chen et al. [1] has not been repeated by any lab.
```

**Early approaches.** Early work on neural network code generation relied on textual similarity metrics for evaluation. For instance, Feng et al. [17] evaluate their CodeBERT model on six programming languages using BLEU [36]. Ren et al. [27] proposes a code generation-specific formulation of this metric. However, previous work has found that textual similarity metrics correlate only weakly with code correctness [1, 2, 27], highlighting the importance of benchmark suites with unit tests.

**Other benchmark formats.** The benchmarks that we translate pair code with comments; some other benchmarks pair natural language descriptions of other kinds. For instance, the CoNaLa [11] benchmark consists of matching natural language questions and code snippets mined from StackOverflow. We note that MCoNaLA [37], which extends CoNaLa to Spanish, Japanese, and Russian, is the only currently available benchmark for evaluating code generation from multiple natural languages.

CodeGen [3] introduces a multi-turn benchmark that involves interleaving prompts and results. We believe it could be translated to other languages with some effort, using a variation of the MultiPL-E approach.

InCoder [4] supports "in-filling" or "fill-in-the-middle" and presents a number of infilling tasks, of which the most general is to fill in an arbitrary line of code in the middle of a function. This article does not benchmark in-filling, since the other two models that we evaluate do not support it. However, MultiPL-E-generated solutions are used as multi-language infilling benchmarks for the SantaCoder [34] code generation models, which do support in-filling.

MathQA [2] is benchmark of math word problems, where answers are number-valued expressions in Python and a DSL. Since the canonical solutions are number-valued, it is easy to check that a solution in another lanugage produces the same number, within some epsilon. However, MathQA requires some formulas to be translated for either fine-tuning or few-shot prompting.

**Other monolingual benchmarks.** There are monolingual code generation benchmarks in languages beyond Python. Kulal et al. [9] presents a C++ dataset consisting of crowd-sourced descriptions of lines of code. Iyer et al. [38] present a Java benchmark taken from online code repositories. Zhong et al. [29] and Yu et al. [39] propose benchmarks for SQL. However, we do not believe SQL is amenable to translation, since conventional types in programming languages do not naturally translate to the types of relational tables. Moreover, of these datasets, only Kulal et al.'s includes unit tests to enable evaluation of code correctness [9].

Our approach could be applied to other Python code generation benchmark suites like MathQA-Python [2], a set of mathematical word problems with multiple choice answers, or APPS [10], a set of problems taken from open-access code competition websites like Codeforces.

**Other tasks.** Although we focus specifically on benchmarks for the code generation task, there are many other tasks that have been used to evaluate code generation models, including generating unit tests from code [19], code search [17, 21], and type inference [22, 23, 24]. Lu et al. [20] propose a suite of evaluation datasets for ten tasks, including code translation, docstring generation, and code summarization.

**Other code generation models.** We evaluate three state-of-the-art code generation models, but many other models that have been proposed. Two influential early models were CodeBERT [17] and PyMT [18]. More recent models include PolyCoder [7], CodeParrot [40], AlphaCode [41], and PaLM-Coder [42]. PolyCoder was not trained on natural language text, and its authors explicitly state that it may not be suitable for NL2Code. AlphaCode and PaLM-Coder are not available for academic researchers to investigate.

**Other multi-language evaluation.** Xu et al. [7] measure the performance of several code generation models on 12 languages. However, they evaluate model performance using perplexity, rather than building a benchmark with unit tests, as we do; they test code correctness only for Python.

HumanEval-X[14] is an unpublished benchmark that appeared after our work that manually translates the HumanEval problems into four languages (C++, Java, JavaScript, and Go).

MBXP [43] is a concurrent effort to evaluate code generation models. We support more languages (13 vs. 19), though MBXP translates an additional benchmark (MathQA). Both MBXP and our work could be extended to support more languages and benchmarks. However, there are deeper differences in the nature of our translation and evaluation:

- We believe our approach to testing is more reliable. Rather than keeping the unit tests hidden from the model, MBXP prompts the model with the same unit tests it uses to test the generated code. Thus the code generation model can "see" the test cases that it will be evaluated on. In contrast, we use a hidden set of unit tests to evaluate code correctness.
- Our work is more faithful in translating types from Python into typed languages. For example, our type inference infers types like `Either[X,Y]` and `Optional[X]` and translates them to algebraic datatypes in typed languages (§6.3). MBXP produces types such as `Object` and `Any` in languages like Java and Scala, which are less idiomatic. For languages that do not support `Any`, such as C++, MBXP fail to translate these benchmarks altogether.
- MBXP uses *greedy decoding* in their evaluation of public models. Greedy decoding produces a single candidate program which may not be the most likely program. Prior work has shown that sampling the output of a code generation model significantly improves the correctness of generated code [1]. We follow standard practices for sampling (§2.2).
- Finally, MBXP has publicly released a subset of their benchmarks, but not their system used to build them. All code and data for MultiPL-E is open source.

## 9 Conclusion

We propose MultiPL-E, the first massively parallel, multi-language benchmark for natural-language-to-code generation. We write compilers to translate code generation benchmarks from Python to 18 additional programming languages that span a spectrum of language features and popularity.

14. http://keg.cs.tsinghua.edu.cn/codegeex

We translate two widely used unit test-driven benchmarks for code generation: HumanEval and MBPP. Using our multi-language parallel versions, we present the first multi-language code correctness evaluation of three state-of-the-art models: Codex, CodeGen, and InCoder. We demonstrate that Codex displays high performance across a variety of programming languages, performing similarly to Python on several languages, most notably, JavaScript.

In our detailed by-language analysis, we find a predictable effect of language frequency, but draw mixed conclusions about the impact of type annotations. Our detailed error analysis highlights common patterns in four languages, finding model errors that are both like and unlike those of human programmers. We hope that our in-depth, parallel evaluation of a large set of languages will be a useful guide for developers weighing whether the utility of code generation tools in their project context.

Our publicly available benchmark is also easy to extend to new problems and languages. We hope it will help evaluate and develop future work on multi-language code generation models.

## 10 Acknowledgments

We thank Steven Holtzen and Joydeep Biswas for loaning us their GPUs. We thank Northeastern Research Computing and the New England Research Cloud for providing computing resources. We thank Greg Shlomo and Milson Munakami for technical support with these computing sources. This work was partially supported by the National Science Foundation grant CCF-2052696.

## References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] 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," *arXiv preprint arXiv:2108.07732*, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[3] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," 2022. [Online]. Available: https://arxiv.org/abs/2203.13474

[4] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022. [Online]. Available: https://arxiv.org/abs/2204.05999

[5] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang, M. Pieler, U. S. Prashanth, S. Purohit, L. Reynolds, J. Tow, B. Wang, and S. Weinbach, "Gpt-neox-20b: An open-source autoregressive language model," *arXiv preprint arXiv:2204.06745*, 2022. [Online]. Available: https://arxiv.org/abs/2204.06745

[6] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint arXiv:2203.13474*, 2022. [Online]. Available: https://arxiv.org/abs/2203.13474

[7] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[8] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 21–29.

[9] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. S. Liang, "Spoc: Search-based pseudocode to code," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf

[10] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021. [Online]. Available: https://arxiv.org/abs/2105.09938

[11] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 476–486. [Online]. Available: https://doi.org/10.1145/3196398.3196408

[12] R. Alur, R. Bodik, G. Juniwal, M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2013.

[13] S. Chaudhuri, K. Ellis, O. Polozov, R. Singh, A. Solar-Lezama, and Y. Yue, "Neurosymbolic Programming," *Foundations and Trends in Programming Languages*, vol. 7, no. 3, pp. 158–243, 2021.

[14] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[15] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[16] B. Wang and A. Komatsuzaki, "Gpt-j-6b: A 6 billion parameter autoregressive language model," 2021. [Online]. Available: https://github.com/kingoflolz/mesh-transformer-jax

[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model

for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[18] C. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "PyMT5: multi-mode translation of natural language and python code with transformers," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online: Association for Computational Linguistics, Nov. 2020, pp. 9052–9065. [Online]. Available: https://aclanthology.org/2020.emnlp-main.728

[19] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.10297*, 2020. [Online]. Available: https://arxiv.org/abs/2009.05617

[20] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[21] T. Ahmed and P. Devanbu, "Multilingual training for software engineering," in *Proceedings of the 44th International Conference on Software Engineering*. ACM, 2022.

[22] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "LambdaNet: Probabilistic Type Inference using Graph Neural Networks," in *International Conference on Learning Representations (ICLR)*, 2020.

[23] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep Learning Type Inference," in *Fse*, 2018.

[24] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "TypeWriter: Neural Type Prediction with Search-Based Validation," in *Esecfse*, 2020.

[25] I. Drori, S. Zhang, R. Shuttleworth, L. Tang, A. Lu, E. Ke, K. Liu, L. Chen, S. Tran, N. Cheng, R. Wang, N. Singh, T. L. Patti, J. Lynch, A. Shporer, N. Verma, E. Wu, and G. Strang, "A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level," *Proceedings of the National Academy of Sciences*, vol. 119, no. 32, p. e2123433119, Aug. 2022.

[26] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *ICLR*, 2020.

[27] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

[28] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 3911–3921. [Online]. Available: https://aclanthology.org/D18-1425

[29] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," 2017. [Online]. Available: https://arxiv.org/abs/1709.00103

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[31] A. Aghajanyan, B. Huang, C. Ross, V. Karpukhin, H. Xu, N. Goyal, D. Okhonko, M. Joshi, G. Ghosh, M. Lewis, and L. Zettlemoyer, "Cm3: A causal masked multimodal model of the internet," *arXiv preprint arXiv:2201.07520*, 2022. [Online]. Available: https://arxiv.org/abs/2201.07520

[32] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, "The pile: An 800gb dataset of diverse text for language modeling," 2021. [Online]. Available: https://arxiv.org/abs/2101.00027

[33] D. Bates, M. Mächler, B. Bolker, and S. Walker, "Fitting linear mixed-effects models using lme4," *Journal of Statistical Software*, vol. 67, no. 1, pp. 1–48, 2015.

[34] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, L. K. Umapathi, C. J. Anderson, Y. Zi, J. L. Poirier, H. Schoelkopf, S. Troshin, D. Abulkhanov, M. Romero, M. Lappert, F. De Toni, B. G. del Río, Q. Liu, S. Bose, U. Bhattacharyya, T. Y. Zhuo, I. Yu, P. Villegas, M. Zocca, S. Mangrulkar, D. Lansky, H. Nguyen, D. Contractor, L. Villa, J. Li, D. Bahdanau, Y. Jernite, S. Hughes, D. Fried, A. Guha, H. de Vries, and L. von Werra, "Santacoder: don't reach for the stars!" in *Deep Learning for Code Workshop (DL4C)*, 2023.

[35] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "CodeT: Code generation with generated tests," 2022. [Online]. Available: https://arxiv.org/abs/2207.10397

[36] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. USA: Association for Computational Linguistics, 2002, p. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135

[37] Z. Wang, G. Cuenca, S. Zhou, F. F. Xu, and G. Neubig, "Mconala: A benchmark for code generation from multiple natural languages," 2022. [Online]. Available: https://arxiv.org/abs/2203.08388

[38] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," *arXiv preprint arXiv:1808.09588*, 2018.

[39] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," 2018. [Online]. Available:

https://arxiv.org/abs/1809.08887

[40] L. Tunstall, L. von Werra, and T. Wolf, *Natural Language Processing with Transformers*. O'Reilly Media, 2022. [Online]. Available: https://books.google.com/books?id=nTxbEAAAQBAJ

[41] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. d. M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," 2022. [Online]. Available: https://arxiv.org/abs/2203.07814

[42] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022. [Online]. Available: https://arxiv.org/abs/2204.02311

[43] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, S. K. Gonugondla, H. Ding, V. Kumar, N. Fulton, A. Farahani, S. Jain, R. Giaquinto, H. Qian, M. K. Ramanathan, R. Nallapati, B. Ray, P. Bhatia, S. Sengupta, D. Roth, and B. Xiang, "Multi-lingual evaluation of code generation models," 2022. [Online]. Available: https://arxiv.org/abs/2210.14868

[44] T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. D. Iii, and K. Crawford, "Datasheets for datasets," *Communications of the ACM*, vol. 64, no. 12, pp. 86–92, 2021.