

---

# InCoder: A Generative Model for Code Infilling and Synthesis

---

Daniel Fried<sup>\*♡†◇</sup>   Armen Aghajanyan<sup>\*♡</sup>   Jessy Lin<sup>♣</sup>  
Sida Wang<sup>♡</sup>   Eric Wallace<sup>♣</sup>   Freda Shi<sup>△</sup>   Ruiqi Zhong<sup>♣</sup>  
Wen-tau Yih<sup>♡</sup>   Luke Zettlemoyer<sup>♡†</sup>   Mike Lewis<sup>♡</sup>  
Facebook AI Research<sup>♡</sup>   University of Washington<sup>†</sup>  
UC Berkeley<sup>♣</sup>   TTI-Chicago<sup>△</sup>   Carnegie Mellon<sup>◇</sup>  
dfried@andrew.cmu.edu, {armenag,mikelewis}@fb.com

## Abstract

Code is seldom written in a single left-to-right pass and is instead repeatedly edited and refined. We introduce INCODER, a unified generative model that can perform program synthesis (via left-to-right generation) as well as editing (via infilling). InCoder is trained to generate code files from a large corpus of permissively licensed code, where regions of code have been randomly masked and moved to the end of each file, allowing code infilling with bidirectional context. Our model is the first generative model that is able to directly perform zero-shot code infilling, which we evaluate on challenging tasks such as type inference, comment generation, and variable re-naming. We find that the ability to condition on bidirectional context substantially improves performance on these tasks, while still performing comparably on standard program synthesis benchmarks in comparison to left-to-right only models pretrained at similar scale. The INCODER models and code are publicly released.<sup>2</sup>

## 1 Introduction

Large language models trained on vast repositories of code have demonstrated remarkable progress in neural program synthesis and related tasks [17, 9, 67, 45, 20]. However, such models generate code left-to-right, which makes them less directly applicable to many ubiquitous code editing tasks, such as fixing bugs, adding comments, or re-naming variables. We introduce INCODER, a unified model for program synthesis and editing. Like prior work, INCODER is trained to maximize the likelihood of a corpus of code. However, we adopt a causal masking objective [2], allowing INCODER to infill blocks of code conditioned on arbitrary left and right contexts.

More specifically, we learn to infill by randomly replacing spans of code with a sentinel token and moving them to the end of the sequence (Figure 1, top). The model is trained to predict all tokens in the complete sequence in this permuted ordering. During inference, we can edit code by replacing spans with sentinel tokens, prompting the model with the new sequence, and having it generate new tokens to replace the masked spans (Figure 1, bottom). Because the model can also trivially generate without sentinel tokens, the result is a unified approach for both program synthesis (via left-to-right generation) and editing (via infilling).

We evaluate performance on a range of zero-shot code infilling tasks, both new and from existing work, including challenging use cases such as type prediction, variable re-naming, comment generation,

---

<sup>\*</sup>Equal contribution

<sup>2</sup><https://sites.google.com/view/incoder-code-models>

## Training

Original Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Masked Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        <MASK:0> in word_counts:
            word_counts[word] += 1
        else:
            word_counts[word] = 1
    return word_counts
<MASK:0> word_counts = {}
    for line in f:
        for word in line.split():
            if word <EOM>
```

## Zero-shot Inference

Type Inference

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Variable Name Prediction

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_count = {}
        for line in f:
            for word in line.split():
                if word in word_count:
                    word_count[word] += 1
                else:
                    word_count[word] = 1
    return word_count
```

Docstring Generation

```
def count_words(filename: str) -> Dict[str, int]:
    """
    Counts the number of occurrences of each word in the given file.

    :param filename: The name of the file to count.
    :return: A dictionary mapping words to the number of occurrences.
    """
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Multi-Region Infilling

```
from collections import Counter

def word_count(file_name):
    """Count the number of occurrences of each word in the file."""
    words = []
    with open(file_name) as file:
        for line in file:
            words.append(line.strip())
    return Counter(words)
```

Figure 1: At training time (top), our causal masking objective samples one or more spans of code in training documents (in the upper left figure, a single span) and moves these spans to the end of the document, with their original location denoted by special mask sentinel tokens. An autoregressive language model is trained to produce these entire masked documents, allowing it to learn to generate insertion text conditioned on bidirectional context. At inference time (bottom), we can perform a variety of code editing and infilling tasks in a zero-shot fashion by inserting mask tokens at desired locations and allowing the model to generate code to insert there. All examples shown are real outputs from our INCODER-6.7B model, with the regions inserted by the model highlighted in orange.

and completing missing lines of code. Zero-shot infilling with bidirectional context substantially outperforms approaches based on left-to-right-only models, and on several tasks obtains performance comparable to state-of-the-art models fine-tuned on the tasks. INCODER also achieves strong performance on existing left-to-right program synthesis tasks, where it matches existing code language models trained at similar scale despite its more general training objective.

To enable further work on code infilling and generation, we publicly release our models at <https://sites.google.com/view/incoder-code-models>.

## 2 Infilling and Synthesis via Causal Masking

Neural models for code generation have either utilized a left-to-right (causal) autoregressive language modeling objective [15, 17] or, as BERT does, a masked language modeling objective [21, 23]. Both approaches have strengths and weaknesses. Causal models only condition on context to the left of the

generated tokens, thus preventing infilling, but they can autoregressively generate entire documents. On the other hand, masked language models can condition on both the left and right contexts to infill a masked region, however, their training objective is typically limited to generating only about 15% of a document. In this paper, we adopt the recently proposed *causal masking* objective [2], which aims to combine the strengths of both causal and masked language models.

## 2.1 Training

At training time, the causal masking procedure samples a number of *spans* of contiguous tokens in each document to mask (Figure 1, top left). We sample the number of spans from a Poisson distribution with a mean of one, truncated to the support  $[1, 256]$ , so that there are typically a small number of spans (with a single span around 50% of the time), but the distribution has a long tail (up to 256 spans). The length of each span is sampled uniformly from the length of the document and the set of sampled spans is rejected and resampled if any spans overlap.

Once spans are sampled, each span  $k$  is replaced with a special *mask sentinel token*,  $\langle \text{Mask}:k \rangle$ . The sequence of tokens in the span is then moved to the end of the document (Figure 1, top right), with the mask sentinel token prepended and a special *end-of-mask* token  $\langle \text{EOM} \rangle$  token appended. In other words, when a mask token appears for the first time in the left-to-right ordering, it marks the location the span was removed from; when it appears for the second time, it marks the start of the moved span text. More formally, assume we have a document  $D$  with  $N$  tokens, and we have sampled one span  $\text{Span} = D_{i:j}$ . Let  $\text{Left}$  be the left context  $D_{0:i}$  and  $\text{Right}$  be the right context  $D_{j:N}$ . Then, we maximize the log probability of the masked document:

$$\log P([\text{Left}; \langle \text{Mask}:0 \rangle; \text{Right}; \langle \text{Mask}:0 \rangle; \text{Span}; \langle \text{EOM} \rangle]) \quad (1)$$

where  $;$  denotes sequence concatenation. If more than one span were sampled, each would be similarly appended at the end of the document in order. As in standard left-to-right generative language modeling, we compute the probability of the sequence auto-regressively and train the model using cross-entropy loss on all tokens except the mask sentinel tokens  $\langle \text{Mask}:k \rangle$ , so that the model does not generate these tokens during inference.

## 2.2 Inference

During inference, the model can either be used for left-to-right generation in the standard way (by sampling autoregressively from the model, without using any special tokens), or it can insert code at arbitrary locations in an existing document by inserting a  $\langle \text{Mask}:k \rangle$  tokens at the desired location(s) and continuing generation at the end of the document. Assuming for simplicity of notation that we want to insert text at only a single location, we generate a span to insert between the location’s  $\text{Left}$  and  $\text{Right}$  context sequences by sampling tokens autoregressively from the distribution

$$P(\cdot \mid [\text{Left}; \langle \text{Mask}:0 \rangle; \text{Right}; \langle \text{Mask}:0 \rangle]) \quad (2)$$

until either an  $\langle \text{EOM} \rangle$  token is generated or a task-dependent stopping criterion is achieved.<sup>3</sup>

More generally, when inserting at multiple locations, we condition on the document with multiple mask sentinel tokens inserted and a final mask token appended. For example, to insert at two locations we use  $[A; \langle \text{Mask}:0 \rangle; C; \langle \text{Mask}:1 \rangle; E; \langle \text{Mask}:2 \rangle]$  and infill the masks in order, appending the appropriate  $\langle \text{Mask}:k \rangle$  sentinel tokens to signal the start of generation for the next span, i.e., the completed document for two insertion locations is represented by  $[A; \langle \text{Mask}:0 \rangle; C; \langle \text{Mask}:1 \rangle; E; \langle \text{Mask}:2 \rangle; \langle \text{Mask}:0 \rangle; B; \langle \text{EOM} \rangle; \langle \text{Mask}:1 \rangle; D; \langle \text{EOM} \rangle]$ , where regions  $B$  and  $D$  have been infilled.

When applied to code, this allows us to perform tasks that benefit from the bidirectional context in a zero-shot fashion, as shown in Figure 1, bottom. For example, we can perform Python docstring generation conditioned on both the left context (function signature) and right context (function

<sup>3</sup>In practice, we sample from the distribution  $P(\cdot \mid [\text{Left}; \langle \text{Mask}:0 \rangle; \text{Right}; \langle \text{Mask}:1 \rangle; \langle \text{Mask}:0 \rangle])$ , where we insert an artificial  $\langle \text{Mask}:1 \rangle$  token. Not inserting  $\langle \text{Mask}:1 \rangle$  gives an implicit size hint to the model that the  $\langle \text{Mask}:0 \rangle$  token should be expanded to fill the rest of the 2048 token context window. Instead, inserting a  $\langle \text{Mask}:1 \rangle$  token indicates to the model that some amount of the document is omitted after the right context. We found that including this substantially improved the ability of the model to predict  $\langle \text{EOM} \rangle$  appropriately when generating an infill for  $\langle \text{Mask}:0 \rangle$ . See [2] for more.

implementation). We can also infill multiple interdependent regions, e.g., generate import statements required by a function that the model is generating.

### 3 Training Data

We train our models on a corpus of (1) public code with permissive, non-copyleft, open-source licenses and (2) StackOverflow questions, answers, and comments. Our primary focus in this paper is on the Python language, but we also include code files from 28 total languages and all available StackOverflow content. After the filtering and deduplication steps outlined below, our corpus contains a total of 159 GB of code, 52 GB of it in Python, and a total of 57 GB of content from StackOverflow. See Figure 5 in the Appendix for the total data size by language.

#### 3.1 Code

**Sources.** We obtained code files and repository metadata from GitHub and GitLab through the sites’ public APIs over a period ending on December 9th, 2021. We obtained approximately 670,000 public non-fork repositories which GitHub/GitLab detected as containing primarily Python, JavaScript, or Jupyter Notebook files, and with either an MIT, Apache 2.0, BSD-2, or BSD-3 clause license. We included all code from a list of 28 languages (determined by file extension) contained in these repositories.<sup>4</sup> Since Python files can also be contained in non-majority-Python repositories, we also included all other Python and Jupyter files obtainable through the GitHub archive on BigQuery that we did not already obtain from GitHub directly.<sup>5</sup> We preprocess Jupyter notebooks by including all text and code (with Markdown formatting removed from text cells), with cells demarcated by XML-style tags (see Section 3.3).

**Deduplication.** Recent work has shown that deduplicating training data can improve model performance and reduce the risk of memorizing training data [5, 42, 37]. Our deduplication scheme removes code files using exact match on the sequence of alphanumeric tokens in the file.<sup>6</sup> This removed approximately 75% of the corpus by file size (reducing from 1 TB to 250 GB) as there are numerous duplicated repositories, library dependencies included as source files, and common boilerplate code files (e.g., for Python web frameworks). We also use regular expressions to detect email addresses in the code files and replace them with a dummy address,<sup>7</sup> to reduce the risks of the model memorizing real email addresses or hallucinating fake ones.

**Decontamination.** To ensure that our code generation models can be evaluated on several current code generation benchmarks, we perform data decontamination: removing overlap between our training data and the evaluation sets of these benchmarks. We remove any repositories contained in the validation and test sets of CodeSearchNet [33], as these are used to construct validation and test sets for several of the tasks in CodeXGLUE [44].<sup>8</sup>

**Filtering.** Our filtering is similar to past work on generative models of code [17, 45, 67]: we remove files that contain any line longer than 3000 tokens or an average line length greater than 100 tokens, have less than 40% of their characters being alphanumeric or underscores, or appear to be automatically generated, which we determine using substring match on a small number of phrases

<sup>4</sup>We include source files from C, C++, CSS, C#, Common Lisp, Dart, Forth, Go, HTML, Haskell, Java, JavaScript, Julia, Jupyter, Kotlin, Lua, Matlab, PHP, Perl, Python, R, Ruby, Rust, SQL, Scala, Shell, Swift, and TypeScript, although the great majority of files are Python and JavaScript. See Figure 5.

<sup>5</sup>We use <https://cloud.google.com/blog/topics/public-datasets/github-on-bigquery-analyze-all-the-open-source-code>. We only include repositories with one of the above permissive licenses.

<sup>6</sup>We implement match checking using a Bloom filter [14] whose keys are: the file extension, number of tokens in the file, and an MD5 hash [52] of the sequence of tokens, which is highly accurate at identifying files with exactly matching token sequences.

<sup>7</sup>We replace detected email addresses with `removed@example.com`.

<sup>8</sup>We also search for occurrences of code from the HumanEval [17] dataset using substring match, but we did not find any matches in our training set. The solutions to the problems in this dataset, at the time that we obtained our files from GitHub, were only contained in JSON files. We additionally removed repositories in the evaluation sets of the JulCe tasks [1], although we did not evaluate our model on these tasks in this present work.

produced by automatic code and documentation generation systems.<sup>9</sup> Our decontamination and filtering steps together remove roughly 10% of Python files.

### 3.2 StackOverflow

The second component of our corpus consists of all questions, answers, and comments from StackOverflow. The Pile [25], which was used to train recent generative code models that we compare to in Section 6, also contains these questions and answers but does not contain comments. Qualitatively, we find that comments, together with the infilling ability of the model, allow our model to have some capability to do interactive code editing guided by language (see Figure 11 in the Appendix).

### 3.3 Metadata

We include some metadata on the code files and StackOverflow questions/answers directly in our training data to allow attribute-conditioned generation [39, 74] and attribute prediction. For code file data, our attributes are the code filename, the file extension (as a proxy for language), the file source (GitHub or GitLab), and, for GitHub repositories, the number of stars binned into six buckets.<sup>10</sup> To allow this metadata to be optional when performing left-to-right prompting of the model, we insert each attribute at the beginning of its document with a probability of 50% (allowing the model to learn metadata conditioning); otherwise, we insert it at the end of its document (allowing metadata prediction). See Figure 6a and Figure 6b in the appendix for examples. For StackOverflow, our metadata attributes are the question tags for the topic (e.g., python, django) and the number of votes for each question and answer, binned in the same way as repository stars. We insert comments directly after the questions or answers they were written for. See Figure 6c in the appendix for examples.

### 3.4 Tokenization

To increase the amount of context that our code model can condition on, the length of documents that the model can generate, and the efficiency of training and inference, we train a byte-level BPE tokenizer [55, 50]. We allow tokens to extend across whitespace (excluding newline characters) so that common code idioms (e.g., `import numpy as np`) are represented as single tokens in the vocabulary. This substantially improves the tokenizer’s efficiency—reducing the total number of tokens required to encode our training corpus by 45% relative to the byte-level BPE tokenizer and vocabulary of GPT-2.

## 4 Training

Our primary model is INCODER-6.7B, a 6.7B Transformer [61] language model. We use the same architecture as the dense 6.7B models described in [7]; the Fairseq architecture description can be found in Table 8 in the appendix. INCODER-6.7B was trained on 248 V100 GPUs for 24 days. We perform one epoch on the training data, using each training document exactly once. Our implementation utilized the causal masking implementation [2] available in Fairseq [46], with the underlying library being PyTorch [48]. Our per-GPU batch size was 8, with a maximum token sequence length of 2048. We clip all gradient norms to 1.0 and used the Adam optimizer with  $\beta_1 = 0.9$ ,  $\beta_2 = 0.98$  [40]. For our learning rate scheduler, we use the built-in polynomial decay learning rate scheduler available in [48] with 1500 warmup updates. FairScale was used for improving memory efficiency through fully sharding model states [10].

We compare the validation perplexity of the 6B parameter model and a smaller 1.3B parameter model (see Section 6.2 for details on the training of this 1.3B model) in Figure 2, showing comparable scaling laws to those reported by Aghajanyan et al. [2]. Our models have also not yet saturated and would benefit from further training; we report the performance of the 6.7B model on the HumanEval

---

<sup>9</sup>Our filters target files automatically generated by the Protocol Buffer Compiler, Django, Epydoc, Pdoc, Sphinx, MkDocs, or MathJax.

<sup>10</sup>We size bins using an inverse binary logarithmic scale, so that bucket 0 corresponds to the repositories with star counts up to the 50th percentile, bucket 1 corresponds to the 50th to 75th percentiles, and bucket 5 to those in the 97th percentile and above.

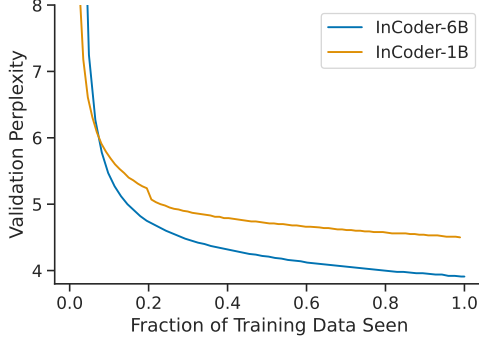


Figure 2: Loss curves show that perplexity is still improving after one epoch and that perplexity improves substantially with a larger model size. This suggests that increasing epochs, data size, or model size would improve performance.

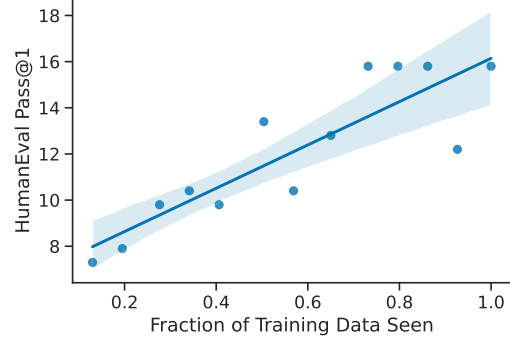


Figure 3: Performance of INCODER-6.7B on the HumanEval left-to-right synthesis benchmark generally increases over the course of pre-training. We plot a line of best fit along with a 95% confidence interval via bootstrap resampling.

Python function synthesis benchmark [17] (see Section 6.1 for a description of this benchmark) and see a consistent increase in performance over the course of training (Figure 3).

## 5 Infilling Experiments

Our primary evaluation is performing zero-shot infilling for a diverse set of programming tasks: filling in multiple lines of code, predicting function return types, generating docstrings, renaming variables, and inserting missing code tokens in a cloze task. We formulate each task as filling in one or more masked-out regions of code.

To evaluate how INCODER benefits from bidirectional context when generating infills, we compare three different inference methods: the causal masking inference procedure described in Section 2, a standard left-to-right generation approach (*left-to-right single*), and a left-to-right generation and reranking approach (*left-to-right reranking*). Since our model is also able to generate left-to-right, we can compare all three inference methods using the same INCODER-6.7B model and thus avoid any confounding effects due to a change in the model. For all three inference methods, we obtain generations from the model using Fairseq’s implementation [46] of top- $p$  (nucleus) sampling [32] with  $p = 0.95$  and a temperature parameter tuned for each task and inference method using the task’s development data. For all generation experiments, we prefix prompts with meta-data indicating the code generated should be Python (i.e., `<| file ext=.py |>`; see Section 3.3).

**Left-to-right single.** This baseline does not use the context to the right of the masked location at all. It generates a single completion for the location by conditioning on the left context and sampling tokens autoregressively from the model  $P(\cdot \mid \text{Left})$  until a task-specific stop condition is reached (e.g., for comment generation, a comment-ending delimiter is produced; for multi-line infilling, a specified number of lines is generated).

**Left-to-right reranking.** This baseline uses only the left context to propose candidates to infill the blank, but uses both the left and right contexts to choose among these candidates. Concretely, we first generate  $K$  possible completions for the blank region,  $\text{Span}_1 \dots \text{Span}_K$  following the same procedure as left-to-right single, using  $K = 10$  unless otherwise specified. We then evaluate each candidate by substituting it into the blank and scoring the completed document. We use either total log probability of the completed document  $\log P([\text{Left}; \text{Span}_k; \text{Right}])$  or, following [17], log probability averaged across the number of tokens in the completed document. We select between these two scoring methods for each task using performance on the task’s development data.

### 5.1 Infilling Lines of Code (HumanEval)

We create an infilling benchmark for complete lines of code from the HumanEval dataset [17]. This dataset provides comment descriptions of functions paired with a canonical implementation of each

Method	Pass Rate	Exact Match	Method	Pass Rate	Exact Match
L-R single	48.2	38.7	L-R single	24.9	15.8
L-R reranking	54.9	44.1	L-R reranking	28.2	17.6
CM infilling	69.0	56.3	CM infilling	38.6	20.6

(a) Single-line infilling.

(b) Multi-line infilling.

Table 1: On our single- and multi-line code infilling benchmarks that we construct from HumanEval, our causal-masked (CM) approach obtains substantial improvements over left-to-right single candidate and left-to-right reranking baselines in both function test pass rate and exact match.

function and several input–output pairs that the function should pass. HumanEval was introduced as a benchmark for the synthesis of entire Python functions; we will evaluate our models on this original synthesis setting in Section 6.1. We use this dataset because it affords functional testing of completed code (as opposed to relying solely on an evaluation of the code surface form), which is particularly important when infilling longer regions that have more potential ways to be completed correctly. We construct two different infilling tasks from the HumanEval dataset, for single-lines and multiple-lines:

**Single-line infilling.** In this task, we mask out each non-blank line of code in the canonical function implementation in turn (creating  $N$  examples for a function with  $N$  non-blank lines). The task is to generate a single-line completion for the blank conditioned on the natural language description of the function and the code lines before and after the blank. We evaluate using (1) pass rate: the rate at which the completed function passes all of the function’s input–output pairs (i.e., analogous to the pass@1 metric from Chen et al. [17]) and (2) exact match: percentage of times that the completed lines exactly match the masked lines in the canonical implementation. Performance is averaged across all examples generated for all programs in the dataset.<sup>11</sup>

**Multi-line infilling.** This task is constructed in the same way as single-line infilling above but allows each masked region to contain multiple lines of code, creating  $N \times (N + 1)/2$  examples for a function with  $N$  non-blank lines. We again evaluate completions using pass rate and exact match, averaged across all infilling examples.

**Inference details.** To choose when to end the infill produced by our inference methods, we truncate the candidates generated by the left-to-right (L-R) baselines to the actual number of lines in the blanked-out region. For our causal-masked (CM) infilling method, we end the infill when the model generates the <EOM> token. For the L-R single and CM infilling methods, we use a temperature of 0.2. For the L-R rerank method, we use a temperature of 0.8 to sample  $K = 10$  candidates and rescore with the total log probability of the completed function.

**Results.** Table 1 shows the results for the single-line (left) and multi-line settings (right). In both settings, CM infilling improves substantially over the L-R single baseline and the L-R reranking baseline. Note that these results are computed by averaging over all examples, which includes masked regions at all positions in functions (including the beginning, when no left context is available, and end, when no right context is available). Figure 4 shows a finer-grained comparison, where we group examples by the fraction of lines in the canonical function which are contained in the context to the right of the infill. The CM infilling method sees larger improvements over the L-R baselines as more right-sided context becomes available (i.e., when the blanked region occurs earlier in the function).<sup>12</sup>

<sup>11</sup>We also report the performance of OpenAI’s recently-released code-davinci-002 code generation system (<https://openai.com/blog/gpt-3-edit-insert/>), which supports infilling, in Table 10, although since no information about this system’s training procedure or data is currently public, this only serves to help gauge the difficulty of our new task.

<sup>12</sup>For multi-line infilling, performance increases with increasing amounts of right context, as having a large right context implies that fewer lines are available to be removed when constructing infill examples, so the resulting examples are often easier to infill.

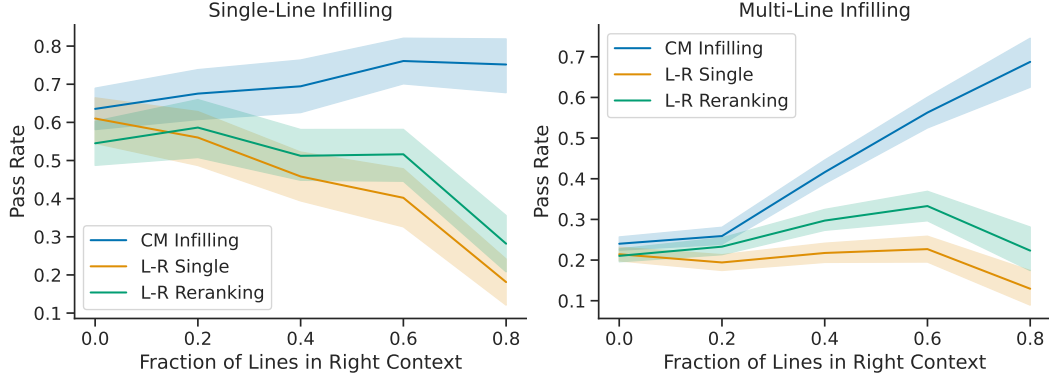


Figure 4: Infilling pass rate by the fraction of the function’s lines which are provided to the right of the region that must be infilled, for single-line infilling (left) and multi-line infilling (right). Shaded regions give 95% confidence intervals, estimated using bootstrap resampling. Our causal-masked (CM) infilling method, blue, consistently outperforms both of the left-to-right (L-R) baselines, with larger gains as more right-sided context becomes available (the right side of both graphs).

## 5.2 Docstring generation (CodeXGLUE)

We next evaluate documentation string (docstring) generation, where models must generate a natural language docstring that summarizes a Python code snippet. Right context may be particularly useful for docstring generation, as conditioning on the function body can allow models to generate more informative descriptions. Prior neural code generation models are fine-tuned on supervised docstring-code pairs to perform this task (e.g., [17, 44, 4]), we evaluate our model zero-shot, with no explicit documentation string supervision.

We use the CodeXGLUE code-to-text docstring generation task [44], which is constructed from CodeSearchNet [33], consisting of docstring-code pairs scraped from publicly available GitHub repositories. The L-R single candidate baseline is prompted with the function signature in the left context preceding the docstring. The CM infilling and L-R reranking methods also observe the right context, consisting of the function body.

We compare models following the original automatic evaluation setup for the task. In Table 2, we report smoothed 4-gram BLEU scores for all models, using the reference docstrings provided in the dataset. These references have been preprocessed to strip extraneous content (e.g., argument definitions) from the original scraped docstrings. We use greedy generation for the CM infilling and L-R single candidate generation methods and sample  $K = 10$  candidates at temperature 0.8 with average log probability scoring for the L-R reranking method (selected by tuning on the validation set of the task). For all inference methods, we terminate generation if the model generates a newline. We also include the performance of the supervised baseline as reported in the CodeXGLUE paper. This baseline is an encoder-decoder model with a CodeBERT encoder fine-tuned on 250k training

Method	BLEU
Ours: L-R single	16.05
Ours: L-R reranking	17.14
Ours: Causal-masked infilling	18.27
RoBERTa (Finetuned)	18.14
CodeBERT (Finetuned)	19.06
PLBART (Finetuned)	19.30
CodeT5 (Finetuned)	20.36

Table 2: CodeXGLUE Python Docstring generation BLEU scores. Our model is evaluated in a zero-shot setting, with no fine-tuning for docstring generation, but it approaches the performance of pretrained code models that are fine-tuned on the task’s 250K examples (bottom block).



Method	Python	JavaScript	Ruby	Go	Java	PHP
Left-to-right single	76.9	77.6	65.8	70.4	74.1	77.1
Left-to-right reranking	<b>87.9</b>	90.1	76.3	92.8	<b>91.7</b>	90.4
CM infill-token	81.8	73.9	81.6	<b>95.4</b>	77.6	87.0
CM infill-region	86.2	<b>91.2</b>	78.9	94.7	89.8	<b>91.4</b>
CodeBERT	82.2	86.4	<b>86.8</b>	90.8	90.5	88.2

Table 3: Accuracy on the CodeXGLUE max/min cloze task. We compare four different inference methods. Left-to-right single: scoring with left-to-right ordering using only the left context and the completion (containing max or min); Left-to-right reranking: scoring with left-to-right ordering using the left context, completion, and right context; CM infill-token: causal masking scoring, using only a single token (containing max or min) as the infill, CM infill-region: causal masking scoring that additionally contains 10 tokens from the right side context.

examples from the dataset. Our zero-shot performance approaches the performance of the fine-tuned CodeBERT model.

### 5.3 Code Cloze (CodeXGLUE)

CodeXGLUE cloze is created from CodeSearchNet to evaluate CodeBERT and consists of a short natural language description followed by code in several programming languages. We evaluate on the max/min subtask, where the model has to decide if the given mask should be filled with either max or min. Since there are only two options in this task, we can closely compare the causal-masked infilling and left-to-right setups by scoring both options and selecting the sequence with the highest likelihood.

Table 3 contains the main results. Using the causal-masked infill format with a single token (containing min/max) as the masked region (CM infill-token) performs better than using just the left context, but not as well as scoring the entire sequence left to right. Masking a larger region (CM infill-region), containing the left prefix and 10 right-side tokens in the masked region, performs comparably to scoring the whole sequence. Infill region length and tokenization can affect the performance, see A.2 for more details and more comparisons.

Note that comparing the scores of the sequences, which differ in their infills, with the left-to-right setup is more computationally expensive than with the CM infilling setup, as the Transformer intermediate activations can be cached and shared across identical sequence prefixes, and in the CM infill setup all sequence differences occur at the ends.

### 5.4 Return Type Prediction

Predicting return type hints for Python functions is a challenging structured generation task (see Figure 1, “type inference”). We evaluate on two datasets: one we construct from CodeXGLUE and the dataset from TypeWriter OSS [49].

**CodeXGLUE.** We develop a benchmark for return type prediction using the same Python CodeXGLUE dataset used in the code-to-text (docstring generation) task. We run an abstract syntax tree (AST) processor and transformer on all functions in the development and test sets of this dataset to (1) identify functions with a PEP 484<sup>13</sup> return type hint annotation that is not None and (2) remove all other type hints (e.g., for function arguments and variable declarations) from the function. This leaves 232 functions in the development and 469 functions in the test set.

The task is to condition on the function signature and body and predict the type hint. We compare the type hints predicted by our various methods to the annotated type hint in the original function, using exact match accuracy on the normalized type hint<sup>14</sup>

<sup>13</sup><https://peps.python.org/pep-0484/>

<sup>14</sup>We normalize each type hint by first parsing the type to an AST and then un-parsing the AST to a surface form string, then compute exact match on these surface forms. We note that this metric is somewhat noisy, given

Method	Accuracy	Method	Precision	Recall	F1
Left-to-right single	12.0	Ours: Left-to-right single	30.8	30.8	30.8
Left-to-right reranking	12.4	Ours: Left-to-right reranking	33.3	33.3	33.3
Causal-masked infilling	<b>58.1</b>	Ours: Causal-masked infilling	<b>59.2</b>	<b>59.2</b>	<b>59.2</b>
(a) Results on the test set of the benchmark that we construct from CodeXGLUE.		TypeWriter (Supervised)	54.9	43.2	48.3
		(b) Results on a subset of the TypeWriter’s OSS dataset [49]. We include examples from which we were able to obtain source files, successfully extract functions and types, that have non-None return type hints, and that were not included in our model’s training data.			

Table 4: Results for predicting Python function return type hints on two datasets. We see substantial improvements from causal masked infilling over baseline methods using left-to-right inference.

To compare our three generation methods, we stop generation when a `:` is generated, which ends the type hint and signals the start of the function body. We tune inference hyperparameters on the development set, and we use a temperature of 0.2 for left-to-right-single, 0.8 for left-to-right reranking, and greedy generation for causal masked infilling. Results on the test set are given in Table 4a. Conditioning on the right context (i.e., the function body) gives some benefit in the left-to-right reranking setting, but gives a substantial improvement via our causal masked infilling.

**TypeWriter OSS.** Some recent work has developed supervised machine learning approaches for predicting type annotations for dynamically-typed languages including Python [68, 6, 49] and TypeScript [30, 65, 35]. We compare our zero-shot model to one such approach for Python, TypeWriter [49], which combines a neural architecture for type hint prediction with a search-based incremental type validation procedure.

To compare to the supervised TypeWriter approach, we obtain its predictions on the open-source software (OSS) dataset used in that work [49], consisting of Python functions from GitHub. Unfortunately, we could not evaluate on their full evaluation set since much of it was included in our model’s training data. We filter to instances that were not included in our training data, for which we were able to obtain files and extract functions and types from via AST parsing, and which have non-None return type hints. This leaves 2,092 examples (about 12% of their evaluation set). We otherwise emulate their exact setup, which allows our model to condition on file imports, the function body, and the function signature to predict return type hints. We use the same inference hyperparameters as we did for CodeXGLUE type hint prediction.

We present our results in two tables: Table 4b which represents metrics across non-None types, and Table 11 in the Appendix, which includes None types as well (following [49]).<sup>15</sup> We again see substantial benefits from causal masked infilling’s ability to condition on the function body when generating return types, and we find that our zero-shot model outperforms the supervised TypeWriter model on this dataset.

## 5.5 Variable Name Prediction

Variable name prediction is a less-constrained code generation task that requires modeling bidirectional context. We again use the test set from the CodexGlue code-to-text task (docstring generation) and run an AST transform to isolate and either mask all the occurrences of the variable name (infilling) or take the left-most context from the first variable name (left-to-right mode). In the infilling setting, given that we generate the number of masks equivalent to the number of times a variable is seen, we select the most common prediction as our singular prediction. Furthermore, we only evaluate the set of variable names containing four or more characters. For our re-ranking, we consider a candidate set of 25 variables. We present our results in Table 5. We again see substantial benefits from using both left and right context: left-to-right reranking and causal-masked infilling both outperform the

that human-annotated type hints can be inaccurate, and that exact match does not reason about type unification or equivalence (e.g., there is no partial credit given for predicting `Optional[str]` rather than `Union[None, str]`).

<sup>15</sup>Our model makes a prediction for every example, so it has identical precision, recall, and F1 scores.

Method	Accuracy
Left-to-right single	18.4
Left-to-right reranking	23.5
Causal-masked infilling	30.6

Table 5: Results on the variable renaming benchmark that we construct from CodeXGLUE. Our model benefits from using the right-sided context in selecting (L-R reranking and CM infilling) and proposing (CM infilling) variable names.

left-to-right single baseline (which uses only the left context). Causal-masked infilling substantially on the left-to-right reranking method, demonstrating the value of conditioning on the right context when proposing candidate completions.

## 6 Model Comparisons

In this section, we compare our model to other generative code models. All published generative models of code, to our knowledge, have been left-to-right only.<sup>16</sup> We compare existing models on two left-to-right Python program synthesis benchmarks in Section 6.1, finding that our model obtains similar performance to models of similar size trained on similar amounts of data. We also perform model ablations and variations of our loss, training data, and model size in Section 6.2. For example model outputs (both for infilling and left-to-right generation), see Section A.4.

### 6.1 Comparison to Left-to-Right Generative Models on Code Synthesis

We compare to past published work on generative code models on the HumanEval [17] and MBPP [9] benchmarks, which require models to condition on natural language descriptions (docstrings) to produce Python programs (typically a single function), and evaluates overall functional accuracy (pass rate) across examples using several test cases for each program.

We evaluate our INCODER-6.7B model in zero-shot evaluation on both of these benchmarks. For HumanEval, we follow past work by prompting with function signatures and docstring descriptions, sample 200 candidate program completions, and compute pass@1, pass@10, and pass@100 using the unbiased sampling estimator of Chen et al. [17]. For MBPP, which does not include function signatures, we prompt only with the docstring description and compute pass@1 [20] using a single candidate.<sup>17</sup> We use top-p sampling with  $p = 0.95$ , with a temperature of 0.2 for pass@1 and 0.8 for pass@10 and pass@100.

We compare our INCODER-6.7B model to models from past work (which have all been left-to-right only) in Table 6, giving the model size and training data summary statistics as reported (or estimated, in cases when a paper only reports token counts, as tokenizer efficiencies vary) in these papers. While differences in details of the Transformer model architectures, datasets, and training procedures across papers and experimental setups make a rigorous comparison impossible, we note that our model achieves roughly comparable performance on the HumanEval metrics to CodeGen-Multi [45], which is also a ~6B parameter model trained on roughly the same amount of Python code, as well as AlphaCode’s 1.1B decoder-only model [43] which also uses a similar amount of Python training data.

### 6.2 Ablation Experiments

For a more rigorous comparison on the effect of training a model with causal masking, and for an analysis of the effects of model size and the training data, we train several variations of our model.

<sup>16</sup>OpenAI has released API access to a system, code-davinci-002, that can perform single-region infilling, but no details have yet been made public about this system, its training data, or any postprocessing done on its outputs. We compare to results obtained from their API on an infilling task in Table 10 in the Appendix.

<sup>17</sup>While this setting is not directly comparable to the three-shot setting where the models of Austin et al. [9] and Chowdhery et al. [20] performed best, we found that our model did not benefit from additional examples in the prompt, which we attribute to much smaller size of our model (6.7B, versus 137B or 540B parameters) and the sensitivity of in-context learning to model scale.

Model	Size (B)	Python Code (GB)	Other Code (GB)	Other (GB)	Code License	Infill?	HE @ 1	HE @ 10	HE @ 100	MBPP @ 1
<i>Released</i>										
CodeParrot [60]	1.5	50	None	None	—		4.0	8.7	17.9	—
PolyCoder [67]	2.7	16	238	None	—		5.6	9.8	17.7	—
GPT-J [62, 17]	6	6	90	730	—		11.6	15.7	27.7	—
INCODER-6.7B	6.7	52	107	57	Permissive	✓	15.2	27.8	47.0	19.4
GPT-NeoX [13]	20	6	90	730	—		15.4	25.6	41.2	—
CodeGen-Multi [45]	6.1	62	375	1200	—		18.2	28.7	44.9	—
CodeGen-Mono [45]	6.1	279	375	1200	—		26.1	42.3	65.8	—
CodeGen-Mono [45]	16.1	279	375	1200	—		29.3	49.9	75.0	—
<i>Unreleased</i>										
LaMDA [9, 59, 20]	137	None	None	???	—		14.0	—	47.3	14.8
AlphaCode [43]	1.1	54	660	None	—		17.1	28.2	45.3	—
Codex-12B [17]	12	180	None	>570	—		28.8	46.8	72.3	—
PaLM-Coder [20]	540	~20	~200	~4000	Permissive		36.0	—	88.4	47.0

Table 6: A comparison of our INCODER-6.7B model to published code generation systems using pass rates @  $K$  candidates sampled on the HumanEval and MBPP benchmarks. All models are decoder-only transformer models. A “Permissive” code license indicates models trained on only open-source repositories with non-copyleft licenses. The GPT-J, GPT-NeoX, and CodeGen models are pre-trained on The Pile [25], which contains a portion of GitHub code without any license filtering, including 6 GB of Python. Although the LaMDA model does not train on code repositories, its training corpus includes ~18 B tokens of code from web documents. The total file size of the LaMDA corpus was not reported, but it contains 2.8 T tokens total. We estimate the corpus size for PaLM using the reported size of the code data and the token counts per section of the corpus.

We compare model pass@1 scores on the HumanEval [17] and MBPP [9] left-to-right synthesis benchmarks, with results in Table 7.

**Objective.** Comparing 1.3B parameter models trained on the same training data with the causal masked (CM) objective (row 2) and the standard left-to-right language modeling (LM) objective (row 3), we see that the causal-masked model obtains slightly higher performance on the HumanEval and MBPP tasks in pass@1 score. This provides further evidence that causal masking training does not hurt the model’s ability to perform standard left-to-right generation.

**Model size.** With data held constant, increasing model size consistently improves performance (comparing the 6.7B and 1.3B CM models in rows 1 and 2, as well as the 1.3B and 2.3B LM models in rows 3 and 6).

**Effects of data.** We compare models trained on our entire dataset of multiple code languages and StackOverflow (multi lang + SO, described in Section 3.1) to data ablations that train only on Python code files and StackOverflow (Python + SO) and only Python code files (Python). We find that training on multiple languages gives a slight reduction in performance on these Python evaluations. However, comparing rows 4 and 5, we see that including StackOverflow data in training substantially improves performance on both HumanEval and MBPP. This suggests that future work on generative code models for language-guided synthesis tasks should consider using StackOverflow or other corpora that mix natural language and code as training data.

## 7 Qualitative Examples

We show a variety of qualitative examples from our model in Section A.4 in both the infilling and left-to-right generation modes: docstring generation, metadata conditioning, class attribute inference from class usage, comment-conditioned code editing, interactive code editing via natural language descriptions, StackOverflow title and tag generation, and zero-shot bidirectional translation of technical jargon between Chinese and English.

## 8 Related Work

**Language Models for Code** There has been a flurry of recent work on training large-scale neural language models on source code. Existing models differ in their architectural design and training

#	Size (B)	Obj.	Training Data	Data Size	Train Tokens	Train Compute	HumanEval Pass@1	MBPP Pass@1
1)	6.7	CM	multi lang + SO	204 GB	52 B	3.0 Z	16	19.4
2)	1.3	CM	multi lang + SO	204 GB	52 B	0.6 Z	8	10.9
3)	1.3	LM	multi lang + SO	204 GB	52 B	0.6 Z	6	8.9
4)	1.3	LM	Python + SO	104 GB	25 B	0.3 Z	9	9.8
5)	1.3	LM	Python	49 GB	11 B	0.1 Z	5	6.1
6)	2.3	LM	multi lang + SO	204 GB	52 B	1.1 Z	9	12.7

Table 7: Ablation results, comparing model performance on the Python portion of a validation set held out from our training corpora as well as the HumanEval and MBPP benchmarks. We compare models by size (in billions of parameters), objective (causal masked, CM, versus standard left-to-right language modeling, LM), training data, and total amount of compute in training (in zettaflops,  $1 \times 10^{21}$  floating point operations).

objectives, e.g., decoder-only language models [9, 17, 34, 67, 45], encoder-only masked language models [23, 36], and encoder-decoder models [4, 43, 53, 63]. Decoder-only language models have recently grown in popularity as they can perform zero-shot program synthesis by generating in a left-to-right fashion. On the other hand, InCoder is a decoder-only causally-masked language model that can infill arbitrary spans of text. This allows the model to perform zero-shot program synthesis and a myriad of other code infilling tasks.

**Infilling Models** Many real-world applications require infilling sequences using left and right context, e.g., editing sentences [56], restoring ancient text [8], and fixing bugs in source code. Unfortunately, standard left-to-right language models cannot directly infill text, and popular masked language models are mainly trained to infill very short spans [16, 21, 51, 53]. Recent work addresses this by changing model architectures, inference procedures, and training objectives [2, 58, 66, 3]. Most related to our approach is the work of Donahue et al. [22] and CM3 [2], who train left-to-right language models to fill in masked token segments of varying lengths. In addition, concurrent to our work, OpenAI released an API<sup>18</sup> that enables code and text infilling, although their models are not open-sourced and the technical details are not public.

**Machine Learning for Code Assistance** There is an extensive literature on using machine learning models to aid human programmers. This includes methods to infer variable types [49, 65], generate unit tests [24], repair programs [29, 69, 19, 70], and verify program correctness [54]. Our InCoder model can infill arbitrary spans of code, which allows it to naturally complete many of the above tasks in one unified zero-shot approach.

**Machine Learning for Program Synthesis** Program synthesis approaches directly generate programs from a specification of functionality [28]. Such models work by taking e.g., input-output examples [11, 27, 18, 12], partial implementations [57], or natural language descriptions [73, 72, 71, 41, 17] of the desired program as input. Our InCoder model differs from this past work as it can both synthesize and infill arbitrary spans of code, conditioning on natural language as well as partial implementations.

## 9 Conclusion

We demonstrated that using a causal masking objective when training a generative model of code enables strong zero-shot performance on many challenging and practical code infilling and editing tasks. The model’s additional infilling capability does not appear to harm its ability to do standard left-to-right generation: ablation and comparison experiments show that our causal-masked models have comparable performance to similarly-resourced models on standard left-to-right language-to-code synthesis benchmarks.

Looking forward, we expect our model performance to continue to increase with more parameters, data, and training steps [38, 31]. Moreover, fine-tuning would allow our models to be better able to

<sup>18</sup><https://openai.com/blog/gpt-3-edit-insert/>

condition on natural language instructions and other indications of human intent [75, 64, 47]. Finally, our model lays a foundation for future work on supervised infilling & editing via model fine-tuning, as well as performing iterative decoding, where the model can be used to refine its own output [26].

## Acknowledgments

We thank the Code Clippy team for open sourcing their data acquisition and deduplication code,<sup>19</sup> which we used as a starting point for our corpus collection. We are also grateful to Srinu Iyer, Marcin Kardas, Stephen Roller, Susan Zhang, Sam Shleifer, Evan Pu, Frank Xu, and Noah Goodman for helpful suggestions and discussions, and to Lucile Saulnier, Leandro von Werra, Nicolas Patry, Suraj Patil, Omar Sanseviero, and others at HuggingFace for help with the model release.

## References

- [1] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. JuIce: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of EMNLP*, 2019.
- [2] Armen Aghajanyan, Bernie Huang, Candace Ross, Vladimir Karpukhin, Hu Xu, Naman Goyal, Dmytro Okhonko, Mandar Joshi, Gargi Ghosh, Mike Lewis, and Luke Zettlemoyer. CM3: A causal masked multimodal model of the Internet. *arXiv preprint arXiv:2201.07520*, 2022.
- [3] Armen Aghajanyan, Dmytro Okhonko, Mike Lewis, Mandar Joshi, Hu Xu, Gargi Ghosh, and Luke Zettlemoyer. HTLM: Hyper-text pre-training and prompting of language models. In *ICLR*, 2022.
- [4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *NAACL*, 2021.
- [5] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *SPLASH*, 2019.
- [6] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *PLDI*, 2020.
- [7] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, et al. Efficient large scale language modeling with mixtures of experts. *arXiv preprint arXiv:2112.10684*, 2021.
- [8] Yannis M. Assael, Thea Sommerschildt, and J. Prag. Restoring ancient text using deep learning: a case study on Greek epigraphy. In *EMNLP*, 2019.
- [9] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- [10] Mandeep Baines, Shruti Bhosale, Vittorio Caggiano, Naman Goyal, Siddharth Goyal, Myle Ott, Benjamin Lefaudeux, Vitaliy Liptchinsky, Mike Rabbat, Sam Sheiffer, Anjali Sridhar, and Min Xu. FairScale: A general purpose modular PyTorch library for high performance and large scale training. <https://github.com/facebookresearch/fairscale>, 2021.
- [11] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *ICLR*, 2017.
- [12] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. AutoPandas: neural-backed generators for program synthesis. *PACMPL*, 2019.
- [13] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. GPT-NeoX-20B: An open-source autoregressive language model. 2022.

---

<sup>19</sup><https://github.com/CodedotAI/gpt-code-clippy>

- [14] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.
- [15] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *NeurIPS*, 2020.
- [16] William Chan, Nikita Kitaev, Kelvin Guu, Mitchell Stern, and Jakob Uszkoreit. KERMIT: Generative insertion-based modeling for sequences. *arXiv preprint arXiv:1906.01604*, 2019.
- [17] 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.
- [18] Xinyun Chen, Petros Maniatis, Rishabh Singh, Charles Sutton, Hanjun Dai, Max Lin, and Denny Zhou. SpreadsheetCoder: Formula prediction from semi-structured context. In *ICML*, 2021.
- [19] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Monperrus Martin. SequenceR: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2021.
- [20] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
- [21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.
- [22] Chris Donahue, Mina Lee, and Percy Liang. Enabling language models to fill in the blanks. In *ACL*, 2020.
- [23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. In *EMNLP Findings*, 2020.
- [24] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *ACM SIGSOFT*, 2011.
- [25] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The Pile: An 800GB dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.
- [26] Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. Mask-Predict: Parallel decoding of conditional masked language models. In *EMNLP*, 2019.
- [27] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM SIGPLAN Notices*, 2011.
- [28] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. In *Foundations and Trends in Programming Languages*, 2017.
- [29] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix: Fixing common C language errors by deep learning. In *AAAI*, 2017.
- [30] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. Deep learning type inference. In *ESEC/FSE 2018*, 2018.
- [31] Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B Brown, Prafulla Dhariwal, Scott Gray, et al. Scaling laws for autoregressive generative modeling. *arXiv preprint arXiv:2010.14701*, 2020.
- [32] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *ICLR*, 2020.

- [33] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.
- [34] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. CodeFill: Multi-token code completion by jointly learning from structure and naming sequences. In *ICSE*, 2022.
- [35] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. Learning type annotation: Is big data enough? In *ESEC/FSE*, 2021.
- [36] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *ICML*, 2020.
- [37] Nikhil Kandpal, Eric Wallace, and Colin Raffel. Deduplicating training data mitigates privacy risks in language models. *arXiv preprint arXiv:2202.06539*, 2022.
- [38] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
- [39] Nitish Shirish Keskar, Bryan McCann, Lav R Varshney, Caiming Xiong, and Richard Socher. CTRL: A conditional transformer language model for controllable generation. *arXiv preprint arXiv:1909.05858*, 2019.
- [40] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.
- [41] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. SPoC: Search-based pseudocode to code. In *NeurIPS*, 2019.
- [42] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. In *ACL*, 2022.
- [43] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *arXiv preprint arXiv:2203.07814*, 2022.
- [44] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. CodeXGlue: A machine learning benchmark dataset for code understanding and generation. In *NeurIPS*, 2021.
- [45] 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.
- [46] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. Fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*, 2019.
- [47] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.
- [48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.
- [49] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: Neural type prediction with search-based validation. In *ACM SIGSOFT*, 2020.
- [50] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. Technical report, OpenAI, 2019.



- [51] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. In *JMLR*, 2020.
- [52] Ronald Rivest. RFC1321: The MD5 message-digest algorithm, 1992.
- [53] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. DOBF: A deobfuscation pre-training objective for programming languages. In *NeurIPS*, 2021.
- [54] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Sekhar Jana. CLN2INV: Learning loop invariants with continuous logic networks. In *ICLR*, 2020.
- [55] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *ACL*, Berlin, Germany, 2016.
- [56] Yong-Siang Shih, Wei-Cheng Chang, and Yiming Yang. XL-Editor: Post-editing sentences with XLNet. *arXiv preprint arXiv:1910.10479*, 2019.
- [57] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.
- [58] Mitchell Stern, William Chan, Jamie Kiros, and Jakob Uszkoreit. Insertion transformer: Flexible sequence generation via insertion operations. In *ICML*, 2019.
- [59] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. LaMDA: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.
- [60] Lewis Tunstall, Leandro von Werra, and Thomas Wolf. *Natural Language Processing with Transformers*. O’Reilly Media, Inc., 2022.
- [61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *NeurIPS*, 2017.
- [62] Ben Wang and Aran Komatsuzaki. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. <https://github.com/kingoflolz/mesh-transformer-jax>, May 2021.
- [63] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, 2021.
- [64] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners. In *ICLR*, 2022.
- [65] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. LambdaNet: Probabilistic type inference using graph neural networks. In *ICLR*, 2020.
- [66] Peter West, Ximing Lu, Ari Holtzman, Chandra Bhagavatula, Jena Hwang, and Yejin Choi. Reflective decoding: Beyond unidirectional generation with off-the-shelf language models. In *ACL*, 2021.
- [67] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169*, 2022.
- [68] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *SIGSOFT*, 2016.
- [69] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *ICML*. PMLR, 2020.
- [70] Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *ICML*, 2021.
- [71] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *ACM MSR*, 2018.

- [72] Tao Yu, Rui Zhang, Kai-Chou Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Z Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *EMNLP*, 2018.
- [73] John M Zelle and Raymond J Mooney. Learning to parse database queries using inductive logic programming. In *AAAI*, 1996.
- [74] Rowan Zellers, Ari Holtzman, Hannah Rashkin, Yonatan Bisk, Ali Farhadi, Franziska Roesner, and Yejin Choi. Defending against neural fake news. In *NeurIPS*, 2019.
- [75] Ruiqi Zhong, Kristy Lee, Zheng Zhang, and Dan Klein. Adapting language models for zero-shot learning by meta-tuning on dataset and prompt collections. In *EMNLP*, 2021.

## A Appendix

### A.1 Corpus statistics and model hyperparameters

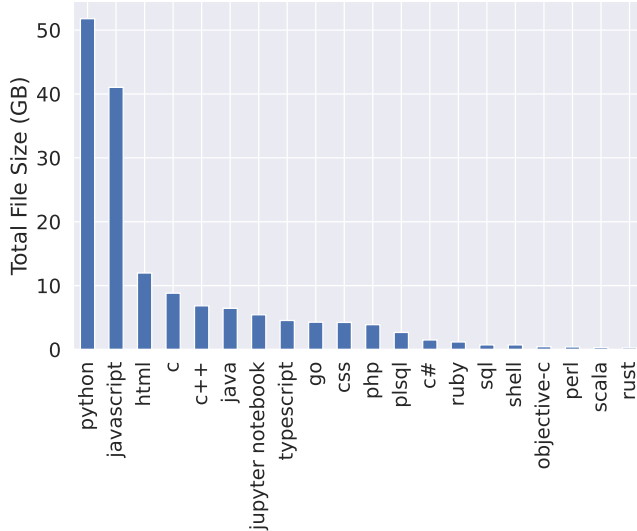


Figure 5: Code corpus composition (after deduplication and filtering) by total file size for the most common languages, as determined by file extension.

Parameter	INCODER-1.3B	INCODER-6.7B
–decoder-embed-dim	2048	4096
–decoder-output-dim	2048	4096
–decoder-input-dim	2048	4096
–decoder-ffn-embed-dim	8192	16384
–decoder-layers	24	32
–decoder-normalize-before	True	True
–decoder-attention-heads	32	32
–share-decoder-input-output-embed	True	True
–decoder-learned-pos	False	False

Table 8: Fairseq architecture hyperparameters for our INCODER models.

### A.2 Cloze and single token infill details

As shown in Table 9, breaking tokenization (-break) on infill decreases the performance using all scoring methods. For example, whereas `Math.max(` was a single token in the full sequence, the sequence is broken into `Math.`, `max`, and `(` for infilling. Infilling with the original tokenization increases the performance slightly, but does not match full left-right scoring. We suspect this is because the model was not trained on infilling single tokens, unlike CodeBERT. A way to fix this is to include a larger region on the left and a few more tokens on the right. This will only slightly increase the scoring complexity. To show that our model uses the right context, we compare it with scoring the left-only model. More precisely, the sequences being scored are

Left-to-right single: [Left; Token]  
Left-to-right reranking: [Left; Token; Right]  
Infill-token: [Left; <Mask:0>; Right; <Mask:1>; <Mask:0>; Token; <EOSS>]  
Left-region: [Left; Token; Right[:10]]  
Infill-region: [ <Mask:0>; Right[10:]; <Mask:1>; <Mask:0>; Left; Token; Right[:10]; <EOSS>]

	Python	Javascript	Ruby	Go	Java	PHP
Left-break	72.4	72.1	68.4	71.7	74.1	76.9
Left-token	76.9	77.6	65.8	70.4	74.1	77.1
Left-region	84.2	88.6	73.7	85.5	87.6	87.0
Left-right-break	77.9	79.4	63.2	89.5	82.0	85.3
Left-right	87.9	90.1	76.3	92.8	91.7	90.4
Infill-break	79.1	83.1	84.2	90.1	84.0	85.3
Infill-token	81.8	73.9	81.6	95.4	77.6	87.0
Infill-region	86.2	91.2	78.9	94.7	89.8	91.4
CodeBERT	82.2	86.4	86.8	90.8	90.5	88.2
Codex*	93.6	93.4	94.7	99.3	95.0	94.3

Table 9: Accuracy on CodeXGLUE cloze max/min. Left: scoring using only the left context, Left-right: score the whole program, Infill: score the infilling sequence, -region: include left context and 10 tokens from the right. -break: break tokenization on the infilled token. Codex\*: version code-davinci-001 of OpenAI’s Codex model, as accessed through their API. Information on the training data for this model is unclear, and it may contain portions of CodeSearchNet (which contains this task’s evaluation set).

Model	Inference	Pass Rate	Exact Match
INCODER-6.7B	Left-to-right single	48.2	38.7
INCODER-6.7B	Left-to-right reranking	54.9	44.1
INCODER-6.7B	Infilling	69.0	56.3
code-davinci-002	Left-to-right single	63.7	48.4
code-davinci-002	Left-to-right reranking	71.8	52.0
code-davinci-002	Infilling	87.4	69.6

Table 10: We evaluate OpenAI’s proprietary code-davinci-002 system, accessed through their API, on our single-line HumanEval infilling task. Although no information is currently public about this system, its training data or procedure, how it performs infills, or whether any postprocessing is done on model outputs, we report its performance to help gauge the difficulty of our new task. For both code-davinci-002 and our INCODER-6.7B model, conditioning on right-sided context improves performance, with the most substantial improvements from infilling.

Method	Precision	Recall	F1
Ours: Left-to-right single	20.0	20.0	20.0
Ours: Left-to-right rerank	24.2	24.2	24.2
Ours: Infill	46.8	46.8	46.8
Ours: Left-to-right single + Return checks	63.2	63.2	63.2
Ours: Left-to-right rerank + Return checks	64.3	64.3	64.3
Ours: Infill + Return checks	76.7	76.7	76.7
TypeWriter (Supervised)	78.8	69.9	74.1

Table 11: Return type hint prediction results on the 25% subset of TypeWriter’s OSS dataset where we were able to obtain source files, extract functions and types from, and that were not contained in our model’s training set. Given an overrepresentation of functions with None in this dataset, and the static analysis capabilities of TypeWriter, we also give results using a simple post-processing step that predicts None if the function does not have any non-trivial return statements.

### A.3 Metadata Examples

```
<| file ext=.py source=github |>
from typing import Dict

def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
</| file filename=string_utils.py dstars=4>
```

(a) Metadata for code includes the file extension, source (github or gitlab), filename, and binned number of stars for GitHub repositories (in logarithmically-sized bins numbered 0 to 5, see Section 3.3).

```
<| file ext=.ipynb:python |>
<text>
This notebook demonstrates using scikit-learn to perform PCA.
</text>
<cell>
%matplotlib inline
</cell>
<cell>
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
</cell>
...
```

(b) In addition to the standard metadata used for all other code files, Jupyter Notebook metadata includes the kernel type (in this instance, Python) as well as the type of the cells in the notebook (either code or text).

```
<| q tags=python,django |>
Django - How to display a form with user input

I'm trying to create a simple web application that can
allow a user to enter a username and password.
Here's what I have so far:
<code>
from django.shortcuts import render
...
</code>
Can you help?
</| q dscore=3 |>
<| c |>
Why aren't you using the django authentication library?
</| c |>
<| a dscore=4 |>
You need to add your form to the template:
<code>
...
</code>
<| a tags=python,django |>
```

(c) Metadata attributes for StackOverflow include question tags and discretized scores of questions and answers.

Figure 6: Examples of metadata attributes included in the training data to allow attribute-conditioned generation and attribute prediction. To allow both generation and prediction, attributes are randomly included either at the beginning of the document or at the end (with probability 0.5 each). Attributes occur in random order to allow arbitrary orderings at inference time.

## A.4 Example Model Outputs

```
def get_conn(self):
    """ """
    conn_config = self._get_conn_params()
    conn = snowflake.connector.connect(**conn_config)
    return conn
```

(a) Reference docstring: Returns a snowflake.connection object.  
Model docstring: Establishes a connection to the Snowflake cluster.

```
def sprint(text, *colors):
    """ """
    return "\33[{}m{content}\33[{}m".format(":".join([str(color) \
        for color in colors, RESET, context=text])) \
        if IS_ANSI_TERMINAL and colors else text
```

(b) Reference docstring: Format text with color or other effects into ANSI escaped string.  
Model docstring: Prints a string with ANSI color codes.

Figure 7: Example docstring generations for the CodeXGLUE code-to-text dataset. Captions for each example give the reference human-written docstring and the output from our INCODER-6.7B model with causal-masked infilling. The model generates docstrings zero-shot by inserting text between the """ comment delimiters.

<pre>&lt;  file ext=.py  &gt; # count the words in all files in the current directory import os import sys  def main():     cwd = os.getcwd()      words = 0      for filename in os.listdir(cwd):         if filename.endswith(".in"):             fname = os.path.join(cwd, filename)             with open(fname) as infile:                 for line in infile:                     words += len(line.split())      print(words)  if __name__ == '__main__':     main()</pre>	<pre>&lt;  file ext=.sh  &gt; # count the words in all files in the current directory find . -type f -name "*.txt" -exec wc -w {} \;   sort -nr   head -n 20</pre>
---	--

Figure 8: Meta-data conditioning on file extensions for Python (left) and Shell (right) allows completing the same text comment as either a Python script or a pipelined bash command, respectively. Regions highlighted in orange are left-to-right generations from our INCODER-6.7B model.

```

class Person:

    def __init__(self, name, age, gender):
        self.name=name
        self.age=age
        self.gender=gender

p = Person('Eren', 18, "Male")

```

Figure 9: Given the beginning of a class definition and right-sided context of the class being used, the model is able to infer plausible attribute names for the class (e.g., “Eren” is likely to be a name, 18 is age, “Male” is the gender.) The region highlighted in orange is an infill generation from our INCODER-6.7B model.

#### Original Code

```

data, target = data.to(device), target.to(device)
optimizer.zero_grad()
output = model(data)
loss = F.nll_loss(output, target)
loss.backward()

data, target = data.to(device), target.to(device)
optimizer.zero_grad()
output = model(data)
loss = F.nll_loss(output, target)
loss.backward()

```

#### 1. Insert comment and regenerate

```

data, target = data.to(device), target.to(device)
optimizer.zero_grad()
output = model(data)
loss = F.nll_loss(output, target)
loss.backward()

data, target = data.to(device), target.to(device)
optimizer.zero_grad()
output = model(data)
# use a Huber loss
criterion = nn.SmoothL1Loss()
loss = criterion(output, target)
loss.backward()

```

Figure 10: By inserting a comment (# use a Huber loss) in the code and replacing the line after with a mask token, a user can prompt the model to in-fill a region of code with a contextually-appropriate replacement. Lines in orange are in-fill generations from our INCODER-6.7B model.

### 1. Generate answer conditioned on question

```
I need to write a Python function called `snake_to_camel`. Here's a description:
"Write a function to convert a snake case string to a camel case string."
<| q tags=python |>
<| a |>
You can use str.title:
<code>
>>> 'my_snake_case'.title()
'MySnakeCase'
</code>
</ a tags=python dscore=3 |>
```

### 2. Add comment

```
I need to write a Python function called `snake_to_camel`. Here's a description:
"Write a function to convert a snake case string to a camel case string."
<| q tags=python |>
<| a |>
You can use str.title:
<code>
>>> 'my_snake_case'.title()
'MySnakeCase'
</code>
</ a tags=python dscore=3 |>
<| c |>
Close, but you need to replace the underscores with an empty string.
<|/ c |>
<| c |>
Thanks, edited!
<|/ c |>
```

### 3. Regenerate infill conditioned on comment

```
I need to write a Python function called `snake_to_camel`. Here's a description:
"Write a function to convert a snake case string to a camel case string."
<| q tags=python |>
<| a |>
<code>
def snake_to_camel(name):
    return ''.join(x.title() for x in name.split('_'))
</code>
</ a tags=python dscore=3 |>
<| c |>
Close, but you need to replace the underscores with an empty string.
<|/ c |>
<| c |>
Thanks, edited!
<|/ c |>
```

Figure 11: Pretraining on StackOverflow, and our model's infilling capability, allows it to perform zero-shot interactive refinement of a function. In the first example, the model has generated the orange region conditioned on the user description. In the second example, the user has added a comment specifying a refinement to the function, and selected the text for the model to replace. In the third example, the orange region has been infilled by the model.



```

<| q tags=tensorflow, gpu |>
How to limit GPU memory usage in TensorFlow?

I am interested in limiting the GPU memory usage in TensorFlow.
I work in an environment in which computational resources are
shared, i.e., we have a few server machines equipped with a few
Nvidia Titan X GPUs each.
For small to moderate size models, the 12 GB of the Titan X is
usually enough for 2-3 people to run training concurrently on
the same GPU. If the models are small enough that a single model
does not take full advantage of all the computational units of
the GPU, this can actually result in a speedup compared with
running one training process after the other. Even in cases
where the concurrent access to the GPU does slow down the
individual training time, it is still nice to have the
flexibility of having multiple users simultaneously train on the
GPU.
The problem with TensorFlow is that, by default, it allocates
the full amount of available GPU memory when it is launched.
Even for a small two-layer neural network, I see that all 12 GB
of the GPU memory is used up.
Is there a way to make TensorFlow only allocate, say, 4 GB of
GPU memory, if one knows that this is enough for a given model?
<|/ q dscore=3 |>

```

Figure 12: Question tag and title prediction from the text of a StackOverflow question. Regions highlighted in orange are infill generations from our INCODER-6.7B model.

```

冒泡排序法 -> bubble sort
动态规划法 -> dynamic programming
快速排序法 -> quick sort
数据结构 -> data structure
词嵌入 -> word embedding

```

Figure 13: Zero-shot bidirectional technical jargon translation between Chinese and English. Regions in orange are infill generations from our INCODER-6.7B model.