A Machine Learning Approach Towards SKILL Code Autocompletion

Enrique Dehaerne[a,b], Bappaditya Dey[b], and Wannes Meert[a]

[a]Dept. of Computer Science, KU Leuven, 3001 Leuven, Belgium
[b]Interuniversity Microelectronics Centre (imec), 3001 Leuven, Belgium

ABSTRACT

As Moore's Law continues to increase the complexity of electronic systems, Electronic Design Automation (EDA) must advance to meet global demand. An important example of an EDA technology is SKILL, a scripting language used to customize and extend EDA software. Recently, code generation models using the transformer architecture have achieved impressive results in academic settings and have even been used in commercial developer tools to improve developer productivity. To the best of our knowledge, this study is the first to apply transformers to SKILL code autocompletion towards improving the productivity of hardware design engineers. In this study, a novel, data-efficient methodology for generating SKILL code is proposed and experimentally validated. More specifically, we propose a novel methodology for (i) creating a high-quality SKILL dataset with both unlabeled and labeled data, (ii) a training strategy where T5 models pre-trained on general programming language code are fine-tuned on our custom SKILL dataset using self-supervised and supervised learning, and (iii) evaluating synthesized SKILL code. We show that models trained using the proposed methodology outperform baselines in terms of human-judgment score and BLEU score. A major challenge faced was the extremely small amount of available SKILL code data that can be used to train a transformer model to generate SKILL code. Despite our validated improvements, the extremely small dataset available to us was still not enough to train a model that can reliably autocomplete SKILL code. We discuss this and other limitations as well as future work that could address these limitations.

Keywords: Automatic programming, Design automation, Neural network applications, Text processing

1. INTRODUCTION

Electronic systems, everything from microwaves to supercomputers, must be designed by hardware engineers. Electronic design is the process of designing an electrical system that meets certain specifications. Electronics can be combinations of connected transistors as well as more complex logical functions, such as processors, controllers, and memory.[1] Electronic design automation (EDA) refers to technologies that automate the design, analysis, and verification of electronic systems. In practice, the purpose of the EDA industry is to provide software tools for hardware engineers to increase their productivity.

Increasing demand and complexity continue to drive innovation in EDA. In 1965, Moore observed that the number of transistors integrated on a chip approximately doubled every two years[2] and predicted that this trend would continue. The ever-increasing complexity of electronic systems requires constant innovation in the EDA industry. One of the first books to advocate for the use of programming languages (PLs) for EDA was *Introduction to VLSI Systems*,[3] published in 1980. Since then, many Hardware Description Languages (HDLs) and other PLs for EDA have been developed and continue to be used today.

SKILL is a PL developed by Cadence[4] that allows hardware design engineers to customize and interact with design software in a programmatic manner.[5] Notably, SKILL can be used for physical layout design[6] as well as printed circuit board design[7] suites. It aims to provide an easy-to-use abstraction layer that is compatible with different underlying design technologies. The creation of SKILL was motivated by the EDA software needs such as computational efficiency and flexibility to underlying design algorithms.[5]

Send correspondence to Enrique Dehaerne (enrique.dehaerne@kuleuven.be)

```
1   /*Create a Pcell that generates rectangles along the x axis which fit within a
2   bounding box with width 15. This pcell accepts the following parameters :
3   width   X - dimension of the rectangle. ( float, default = 1. 0 )
4   length  Y - dimension of the rectangle. ( float, default = 5. 0 )
5   cd1         Space between rectangles in the X - dimension ( float, default = 1. 5 )
6   layer   layer type of the rectangles ( string, default = " metal1 ") */
7
8   pcDefinePCell(
9       list(ddGetObj("pcells") "rectanglesRow" "layout")
10
11      ( (width 1.0) (length 5.0)
12        (cd1 1.5) (layer "metal1") )
13
14      let((tfId bbox)
15          bbox = list(0:0 15:length+1)
16          masterRect = rodCreateRect(
17              ?cvId    pcCellView
18              ?layer   list(layer "drawing")
19              ?width   width
20              ?length  length
21              ?spaceX  cd1
22              ?fillBBox bbox)
23      )
24  )
```
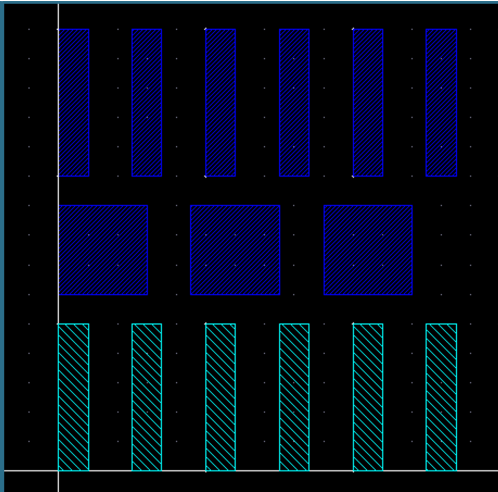
Figure 1. On the left, an example of a parameterized cell (PCell) in the SKILL integrated design environment. The code in red is comments which describe the code underneath it. On the right, instantiations of the PCell with different values of parameters are shown as they appear in the Virtuoso layout editor.[6] The top instantiation uses all default parameter values, the middle instantiation has the height and width values set to 3, and the bottom instantiation has the layer type set to "metal2".

An example SKILL program that describes a parameterized cell (PCell) that can be used to instantiate a (part of a) physical layout is shown on the left side of Figure 1. The code in red is a code comment that describes the PCell and its parameters. When instantiated with the default parameters, the program describes the layout shown in the top right of Figure 1. Below are two instantiations where the values of the parameters differ from the default values.

Machine learning (ML) applied to various steps of the electronic design process has become an important topic of research.[8] Automatic code generation using ML promises to increase the productivity of software developers. Despite a growing body of work in the field of automatic code generation, no work has been published applying this knowledge to SKILL. The transformer[9] neural network architecture has become popular in the field of natural language processing for its efficiency in learning sequences of strings. PLs are often represented as strings called source code and can therefore be processed similarly to natural language (NL), that is, by tokenizing strings into sub-strings and modeling sequences of these tokens. In this way, transformers have been trained on huge code datasets to be able to generate competition-level code at an average human level[10] and for commercial code autocompletion tools.[11]

In this study, a novel methodology for generating SKILL code is proposed. The goal is to use ML methods to generate SKILL code given code contexts with or without NL descriptions (this task is referred to as auto-completion in the rest of this paper) to help make SKILL developers, and hardware design engineers as a result, more productive. The main challenge to overcome towards completing this goal is the extremely limited amount of available SKILL data. To address this problem, our proposed methodology includes:

1. Maximizing the usage of available SKILL code by mining unlabeled and labeled data. In our case, unlabeled data are SKILL code file texts while labeled data are pairs of SKILL code definitions or descriptions and their corresponding bodies. We improve the quality of our dataset through filtering strategies and deduplication.

2. Transfer learning from models pre-trained on large datasets of general PL data, such as Python and Java code. We choose T5-based[12] pre-trained transformers because they are designed for transfer learning and can be fine-tuned using both unlabeled and labeled data.

Additionally, we develop a novel, practical methodology for evaluating synthesized SKILL code. This study provides discussion and experimental results towards reliable SKILL code autocompletion. Experimental results

show that the proposed models outperform similar baseline models. However, limitations of this study, such as the limited available dataset and model size used, contributed to the low overall performance of the proposed models. Therefore the main contributions of this study are extensive discussions of our (i) proposed methodology for curating a custom SKILL code dataset and fine-tuning a pre-trained T5-based model to autocomplete SKILL code, (ii) its experimental results, and (iii) its limitations. Additionally, we suggest future work to improve our results.

The rest of the paper is organized as follows. Next, Section 2 provides an overview of lessons learned from related work in the field of automatic code generation. Section 3 explains the methodology used for (a) creating a custom SKILL dataset (b) training models to generate SKILL code, and (c) evaluating synthesized SKILL code. Section 4 describes the experimental setup and pre-processing steps used for the experiments. Next, Section 5 presents and discusses the experimental results of the proposed methodology compared to baselines. Section 6 discusses limitations of the our study and suggests future work to address these limitations.

2. RELATED WORK

SKILL-related works propose manually written SKILL programs that automate layout design. Tayenjam et al.[13] propose a SKILL PCell to automatically generate inductor layouts given certain input parameters. Abhishek et al.[14] propose an algorithm written in SKILL to optimize inductor layouts toward minimizing loss factors. Searches were conducted in an attempt to find code generation-related work specific to the SKILL PL[5] but none were found. Instead, in the rest of this section, we discuss works related to the automatic generation of HDL code since they are the most similar type of language to the SKILL PL.

Most of the related work on automatic HDL code generation is not ML-based but rather heuristics- or rule-based. The most studied HDLs are Verilog and VHDL. Several research works propose methods that automatically generate Verilog code[15,16] or VHDL code[17] from high-level descriptions. Other research works automatically translate code written in other PLs to Verilog[18,19] or VHDL.[20] These methods that are manually programmed have the advantage that the generated HDL code is guaranteed to be compilable. The disadvantage of these approaches is that they can only generate HDL code for specific functionalities from specific inputs (a list of parameters, code written with specific APIs, etc).

ML-based code generation methods can generate diverse code snippets from a wide variety of input data, for example, previous-written code[21,22] or NL documentation,[23,24] relatively efficiently. However, ML-based code generation methods generally cannot guarantee the compilability or functional correctness of their generated code. Transformer-based language models,[9] that process source code as a sequence of tokens, have become especially popular and effective in code generation tasks.[25] The primary limitation of token-based language modeling is the need for a large amount of source code data and computations to train the models on this data. Most code generation models are trained on huge datasets of code written in general PLs such as Python[22] or Java[26] which are abundantly available on open-source repository databases.[25] State-of-the-art models are usually trained on a dataset of many different PLs, including the target PL, to maximize the amount of code data the model can learn from.[27–29]

To the best of our knowledge, only two previous works have used ML to generate HDL code. Pearce et al.[30] started from the GPT-2 transformer model[31] that was pre-trained on natural language (NL) data and fine-tuned on a synthetic dataset of pairs of English descriptions with Verilog code snippets. Thakur et al.[32] benchmarked recent transformer models pre-trained on general PL code for solving Verilog programming challenges. They fine-tuned the pre-trained models on unlabeled Verilog data from open-source repositories. They manually created a collection of Verilog problems and corresponding test benches to functionally evaluate Verilog code generated by the models.

3. METHODOLOGY

This section introduces the proposed methodology for autocompleting SKILL code from NL documentation and/or previously written SKILL code. As discussed in the previous section, the related literature suggests that transformer ML models are best suited for this problem statement. The main challenge that we had to overcome for this research was the extremely limited amount of available SKILL code data. To overcome this challenge,

we propose fine-tuning models that are pre-trained on general PL code data through both self-supervised and supervised learning on a custom, curated SKILL code dataset. We evaluate our proposed models and similar baseline models using a SKILL-specific BLEU score as well as the SKILL lint tool. Subsections 3.1, 3.2, and 3.3 discuss the custom SKILL dataset, training of the models, and evaluation of the models, respectively.

3.1 Custom SKILL Dataset

A custom SKILL dataset was created because there are no publicly available SKILL code datasets. Although a few proprietary SKILL repositories were available for this study, the volume of code in these datasets was small. Therefore, open-source repositories were mined for additional SKILL programs to be used to train the transformer models. As will be discussed in Section 5.1, the amount of open-source SKILL data was also small and this motivated the use of models pre-trained on large volumes of general PL data.

First, a collection of SKILL source code files was gathered from the proprietary and open-source repositories. These files were then automatically mined to obtain input-output pairs for supervised training and evaluation. An overview of the methodology used to create the custom SKILL dataset is shown in Figure 2. Additionally, three data filtering techniques and one deduplication technique to improve the quality of the training dataset are introduced at the end of this section.
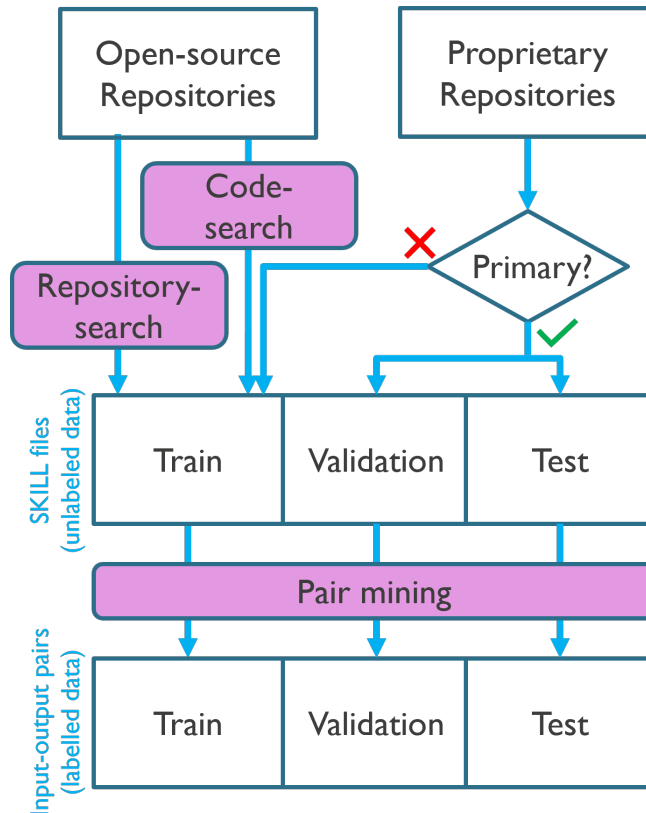

Figure 2. Flowchart showing the sources used and steps taken to create self-supervised and supervised SKILL data. Note that a small amount of data from primary proprietary sources was added to the training split (see Section 3.1.4) which is not depicted in this flowchart.

3.1.1 Proprietary SKILL Data

A variety of proprietary SKILL repositories and resources available to us were gathered for the custom SKILL dataset. This proprietary data is considered to be reliable and high-quality since we can verify that they were developed by SKILL experts at imec. We further distinguish proprietary data into two categories, *primary* and

secondary data. Primary data is data from certain proprietary sources that have desirable qualities for the evaluation of SKILL code autocompletion. We exclusively use primary proprietary data for the validation and test splits of our dataset. Three factors were taken into consideration when classifying a source as primary or secondary.

The first factor was whether the SKILL code was well-documented. The more documentation is available, the more supervised comment-code pairs can be mined out of the data. Code shared between many teams was mostly well documented while those that directly implement solutions (PCells, GUI scripts, etc) were mostly undocumented. An important subset of the code not belonging to one particular team was code with example programs that are used mostly for training SKILL developers.

The second factor was the use of the standard SKILL libraries. This is correlated with the documentation of the files since code shared between teams usually does not use custom libraries while direct implementations do. Adding code contexts from imported libraries to improve the generated code requires inter-file knowledge and linking techniques or advanced code-searching techniques. This is considered out of the scope of this study. That is why, for evaluation purposes, SKILL files that use mostly the standard SKILL libraries were considered to be primary data.

The last factor was the technique used to gather data from a source. Most of the proprietary data were collected simply through zipped directories which required no rule-based extraction or modifications. Data that required rule-based extraction included web-based documentation with program examples. The Selenium WebDriver[33] package for Python[34] was used to mine these types of files. A script with custom rules for the documentation structure was written to extract as many comment-code pairs, saved as SKILL files, from the documentation. A limited number of other SKILL files were modified to make them more homogeneous with the rest of the dataset. These mining rules and code transformations were based on custom heuristics with no guarantee that it works as intended for every sample. Therefore, this data was considered to be secondary data.

3.1.2 Open-source SKILL Data

In addition to the high-quality secondary proprietary data, SKILL code training data was obtained from open-source repositories. GitHub[35] is popular for collecting large volumes of code data.[23, 36–38] Unlike proprietary data, open-source code is not reliably high-quality. Open-source data is therefore only included in the training split of the SKILL dataset, not in the evaluation splits. In Section 3.1.4, we propose filtering strategies to remove poor-quality code from our training data. Using the GitHub application programming interface (API), two types of searches were conducted to retrieve publicly available SKILL code. The first search was a repository search. This search retrieved repositories that contain the query phrase: "cadence skill". Not including "cadence" in the query phrase retrieved many repositories that do not relate to the SKILL language. The repositories were forked and all files with SKILL extensions (*.il* and *.ils*) were added to the custom SKILL dataset.

Not all SKILL repositories contained the aforementioned query phrase which is why code file searches were also conducted. Frequently occurring tokens from the previously collected proprietary and repository search SKILL files were used as query phrases. Since there are relatively strict limitations on the rate of requests free GitHub users can make via the API, a small subset of all these tokens were searched. More specifically, one-fifth of all tokens that were seen more than 10 times, a total of 1436 tokens, were used as queries for GitHub code searches. Additionally, the file extension filter was used to only retrieve *.il* and *.ils* files.

The code file searches retrieved files of other PLs that use the same extensions (e.g., the *Intermediate Language*[39]). A list of blacklisted terms that were found to be popular in non-SKILL files and very rare in SKILL files was used to identify and remove non-SKILL files. The files retrieved were first filtered based on the inclusion of several terms that were frequently found in URLs for non-SKILL files. The remaining files were downloaded and further filtered if they contained any of a list of patterns popular in non-SKILL files that are not-syntactically correct in SKILL. The list of exclusion terms for the URLs and exclusion patterns for code are shown in Table 1.

3.1.3 SKILL Input-Output Pairs

The SKILL file dataset collected from proprietary and open-source repositories was mined to obtain input-output pairs for supervised training and evaluation. Ideally, the inputs give enough information, in the form of comments

Table 1. Keywords and patterns used to find and remove URLs and files, respectively, retrieved from code searches.

Filtering stage	Keywords/Patterns
URL	'dotnet', '-ms', 'microsoft', '.net', 'solaris', 'unity', 'logs', 'www'
File	'.assembly', '.NET', '.class', '.method', '.string', '.float', '.inline' (for all: preceded by at least one whitespace character)

and/or preceding code, which would prompt a SKILL developer to write code similar to the output reference. This section explains the heuristics used to mine such input-output pairs from file-level data.

The granularity level for input-output pairs is an important factor. Public datasets for code generation in other PLs often use function-level samples.[38,40–42] This is because functions encapsulate code into units of functionality. Additionally, they provide well-defined arguments that are expected to be used in the body, giving the models more information to guide code generation. Function definitions and preceding comments are the input and the function body is the output for *comment-function* pairs. When there are no preceding comments, these definition-body pairs, called *function-completion* pairs, are also mined. To provide additional context from long function bodies, the input in a function-completion pair can include the first parts of the function body with the output being the rest of the body. This was done for function with long bodies. The final pair type mined was *comment-code* pairs. The input of a comment-code pair is a comment and the output is code that directly follows the comment and does not define a function output. This code can be a single code statement or a construct, such as a *foreach* loop, which contains multiple statements. Figure 3 shows an example for each of the three pair types.

Figure 3. Three example SKILL programs with annotations showing which parts of the program would belong to the input (green) and output (purple) of a pair. The example on the left is equivalent to the program shown in Figure 1 and can be split into a comment-function pair (see Section 3.1.3). The top-right and bottom-right pairs are function-completion and comment-code pairs, respectively. Note that these SKILL programs were manually written and were not included in the SKILL dataset.

3.1.4 Dataset Filtering, Deduplication, & Split

Different filtering strategies for removing low-quality files and improving learning were experimented with to obtain different data subsets. These strategies were: (i) the file filtering technique used (none, a SKILL lint pass grade, a SKILL lint IQ score greater than or equal to 10, or having more than 1 input-output pair found in

the file) and (ii) whether comments were removed from the files. The SKILL lint tool is a static analysis tool used to measure code quality and compilability. It reads files and assigns an IQ score (out of 100) as well as a pass or fail grade to the file. The pass or fail grade is based on significant syntactic errors while IQ takes into account style and efficiency in addition to syntactic correctness. Not training in a self-supervised manner was also experimented with.

For each data subset that could be obtained by combining the filtering strategies described above, supervised training was either applied or not applied. When applied, whether the pairs were deduplicated was treated as an additional filtering strategy. A pair was considered to be a duplicate of another pair if it appeared within the input and/or output of the other pair. Pairs that are not found within other pairs were called *top-level* pairs. The deduplicated training set only contained top-level pairs. In the rest of this paper, we use the term *training strategy* to denote a possible combination of self-supervised and/or supervised learning with different filtering and deduplication techniques applied to the training dataset.

The validation and testing sets contained a balanced number of comment-function and function-completion pairs (all top-level) as well as comment-code pairs (not all top-level). To achieve a balanced number of each pair type for the test and validation datasets, the following steps were taken:

1. The primary files were randomly partitioned into two sets until a similar number of comment-function pairs were mined from each partition.

2. The number of function-completion pairs was larger than the number of comment-function pairs (n) so n random samples from these pairs were kept and the rest were discarded.

3. The number of top-level comment-code pairs was smaller than n so the comment-code pairs in each split consisted of all top-level comment-code pairs and adding random samples from non-top-level comment-code pairs until n comment-code pairs were obtained.

Note that input-output pairs were used to split files between validation and test splits. This means the files from primary proprietary sources that did not contain any input-output pairs were instead added to the training split.

3.2 Models & Training

Transformer models[9] require large amounts of data to perform well on complex tasks such as code generation. The intuition was that the custom SKILL dataset was too small to train a transformer model from scratch. Therefore, using models with weights pre-trained on general PL code data was hypothesized to result in better final models. The T5[12] framework was designed for transfer learning from large unlabeled datasets to task-specific, labeled datasets. CodeTrans[27] and CodeT5[28] are two state-of-the-art models based on T5[12] that are trained on a variety of different tasks in a variety of different PLs. Checkpoints of both these models trained on large volumes of code in multiple PLs are publicly available in the HuggingFace library.[43] The CodeTrans model is trained on more data than CodeT5 but CodeT5 uses code-specific learning objectives to improve its learning. To get the most out of our extremely limited SKILL dataset, we extended T5's transfer learning approach by fine-tuning these models on both unlabeled and labeled SKILL data.

Two baselines were trained and evaluated as well to compare the results to these proposed models. The first baseline is the original T5 model which was trained on a large collection of NL data. This model is called *T5-NL* in the rest of the paper. The second baseline, *T5-SKILL*, is the original T5 model with random initial weights and is therefore only trained on SKILL data. An advantage of training a model from scratch in this way is that the tokenizer can be adapted to the training data. Therefore, a SentencePiece[44] tokenizer was trained on the training set of the custom SKILL dataset which means the T5-SKILL model's vocabulary is SKILL-specific. SentencePiece[44] is a sub-word tokenization algorithm that is also used for the CodeTrans[27] and T5-NL[12] models. CodeT5 uses a byte-pair-encoding[45] sub-word tokenizer similar to the one used by GPT3.[46] The tokenizers for these models are trained on their respective pre-training datasets and are therefore not SKILL-specific.

The proposed models and both baselines were trained using various training strategies (see Section 3.1.4) as shown in Figure 4. For each model type (CodeTrans,[27] CodeT5,[28] T5-NL,[12] and T5-SKILL[12]) a model is
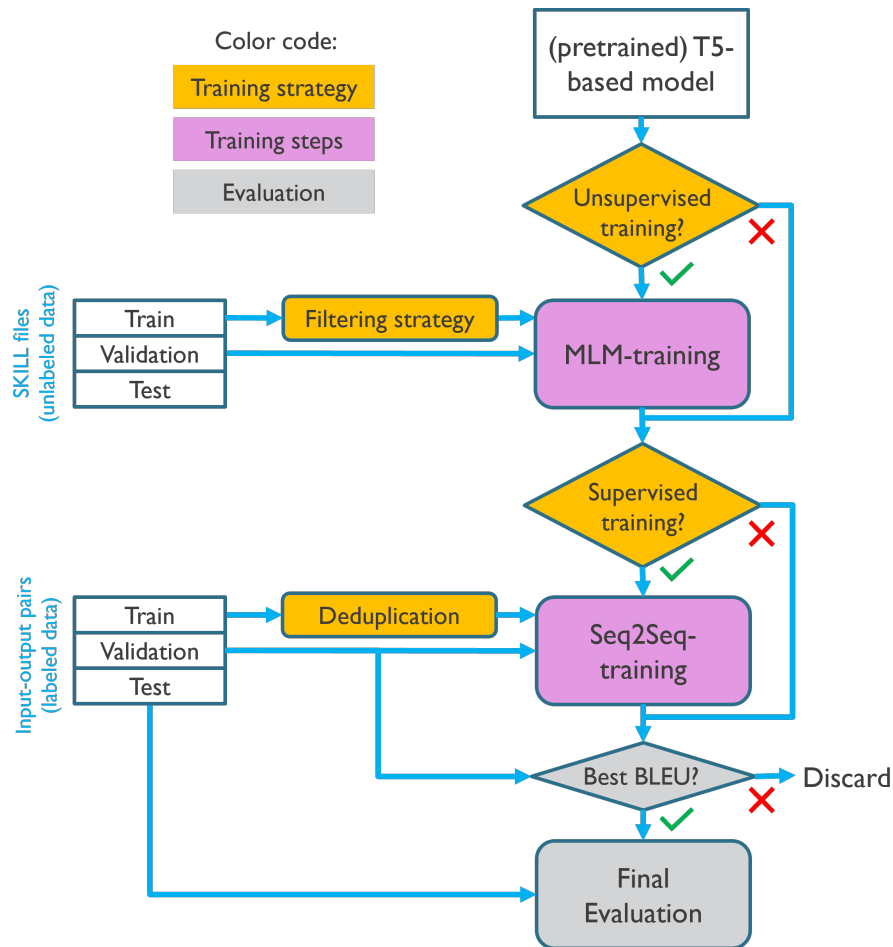
Figure 4. Flowchart showing the high-level training and evaluation steps taken. The "Best BLEU" condition is what decides, for each model type, which training strategy resulted in the best-trained model. This model that achieved the best BLEU score was chosen for final evaluation. Note that certain models that did not achieve the best BLEU score for a given model type were also selected for final evaluation (see Section 5.2).

trained for each of the different possible training strategies. Self-supervised learning is performed using Masked-Language Modeling (MLM) on the SKILL file samples. MLM randomly masks tokens in a token sequence. The task of the ML model being trained is to predict the tokens that have been masked. This allows the model to learn the relationship between tokens before and after the masked tokens. This bi-directional way of modeling is especially useful for encoding the input text which the model generation should be conditioned on. Next, the models are trained in a supervised manner using Sequence-to-Sequence (Seq2Seq) autoregressive modeling on SKILL input-output samples. Seq2Seq training is where the input of an input-output pair is given and the model is tasked with generating the output one token at a time. This uni-directional way of modeling is most useful for the generation procedure since the model only has access to previous tokens it has generated. Figure 5 shows examples of both modeling techniques.

Validation data were used to stop training once validation losses stop improving to avoid overfitting. We have added a sentence to make it more clear. During training, the weights checkpoint with the best validation loss at the end of an epoch was saved. This saved checkpoint is the one used for evaluation on the validation set using the BLEU metric. The training was stopped early after five epochs of no improvement in the validation loss. Models were trained for a maximum of 50 epochs. The best training strategy for each model type in terms of BLEU score[47] was kept for the final evaluation of the test data split. Additional models trained using training strategies that gave surprising results were also selected for final evaluation (see Section 5.2).
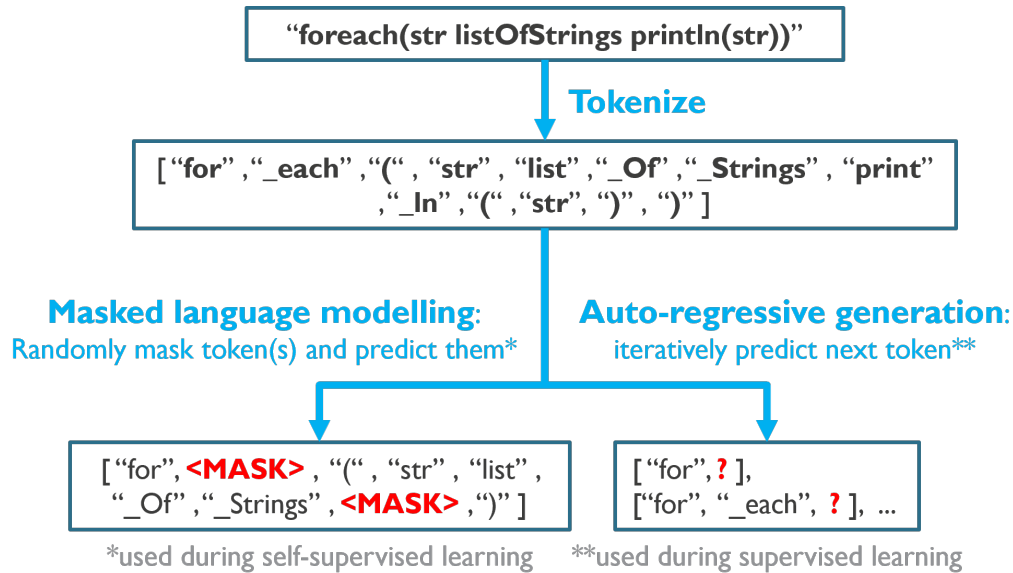
Figure 5. Graphical depiction of MLM and autoregressive modeling for an example SKILL statement.

3.3 Evaluation of Synthesized SKILL Code

One of the most difficult aspects of code generation is evaluation. Other works on code generation[23][32] manually created their programming problems with corresponding verification programs to evaluate their models. While this has the advantage of being able to measure the functional correctness of generated code, it also has two main disadvantages. Firstly, manually creating a functional correctness evaluation dataset is time-consuming. Secondly, manually creating data could introduce bias in the sense that only *toy examples* of problems are written in the evaluation dataset which are not representative of real-world programming tasks. It is very difficult to create a functional correctness dataset from real-world repositories since (i) for a program to run as expected, the dependencies of the original file must be available and (ii) a corresponding verification problem must be available and linked to the code snippet. Instead, we rely on a static-analysis score and the popular BLEU metric[47] to evaluate generated code.

3.3.1 Static-Analysis Metric

The static analysis used in this study was the change (Δ) in the SKILL lint IQ score. As discussed in Section 3.1.4, SKILL lint IQ takes into account the syntactic correctness, style, and efficiency of SKILL programs. For a given input-output pair p and model m, the change in lint IQ score is equal to $liq(f|m(p)) - liq(f)$ with $liq(f)$ the lint IQ score of the file f which p belongs to and $f|m(p)$ the file with the output of p replaced by the model's prediction given the input of p.

3.3.2 BLEU Implementation

The implementation of the BLEU metric used in this study is the geometric mean of the n-gram precisions between the predicted sequence and reference sequences with $n \in \{1, 2, 3, 4\}$. No smoothing was applied. This gives a score between 0 (bad) and 1 (good). Unless mentioned otherwise, the BLEU score was measured using the tokenizer of the respective model being evaluated. That is, the code generated from a model and the output reference code is tokenized using the tokenizer of the generation model and these two sets of tokens are compared. Post[48] argued that using different tokenizers has significant effects on BLEU scores and therefore BLEU scores calculated using different tokenizers were not compared. A standard tokenizer was chosen from the tokenizers of the different model types by measuring the correlation between their BLEU scores and human judgment scores to be able to use BLEU to compare the different model types.

3.3.3 Correlation of Metrics with Human Judgement

A small survey was developed to calculate the correlation between human judgment scores with the change in SKILL lint IQ metric and BLEU calculated with a SKILL-specific tokenizer. For a given input-output pair and output code predictions of all models being evaluated, the survey asked the following questions:

1. "On a scale of 1 (very bad) to 5 (very good), how would you rate the quality of the input prompt (i.e. how well did the input prompt describe the code it was prompting)?"

2. "On a scale of 1 (not at all) to 5 (very much), to what degree does the output reference follow logically from the input prompts?"

3. "On a scale of 1 (very bad) to 10 (very good), rate each model's output based on the quality of the generated SKILL code as well as the degree to which the generated SKILL code logically follows from the input prompt."

Additionally, the survey included open-ended questions for the survey taker to provide any additional feedback they wanted to provide. The survey was sent to SKILL developers who have access rights to view the proprietary data sources of our custom SKILL dataset (see Section 3.1.1).

The survey included 15 SKILL code autocompletion input-output pairs (five of each pair type). The number of input-output pairs was limited to 15 to minimize the time required from human evaluators. The prompts were selected as follows:

1. Randomly sample 60 pairs of the corresponding pair type from the test dataset.

2. Manually select the best 15 samples in terms of code quality and degree to which the output follows logically from the input.

3. Randomly sample 5 of these 15 pairs for inclusion in the survey. The same prompts were used for all different models evaluated in the survey.

This selection procedure was intended to achieve a balance between random sampling and ensuring that non-sensical prompts were not included in the survey. The latter is important due to the small number of pairs and model outputs that could be evaluated.

4. EXPERIMENTAL SETUP & PRE-PROCESSING

All experiments were conducted on a workstation with an NVIDIA GeForce RTX 3070 graphical processing unit (GPU). The HuggingFace library[43] was used for many parts of the experiments including the implementations of the T5[12]-based models*, BLEU[47] metric, and parts of the training and inference scripts. The optimizer used was Adafactor,[49] the same optimizer used for the original T5[12] models. The maximum batch sizes that fit into memory, 11 for self-supervised training and 2 for supervised training, were used.

For file data, duplicate files were removed (the number of files reported in Section 5 is after removing duplicate files). Files were considered to be duplicates if their text values were the same. On the remaining files, unwanted metadata was removed. Unwanted metadata included liability disclaimers, author information (names and emails), and script version information. These types of metadata were found and removed using regular expressions. Characters from foreign characters, as well as other non-ASCII characters, were removed from all the files. Unnecessary comments were observed in the SKILL file dataset. Cases that were removed were

---

*The pre-trained model checkpoints used for each model type were:

- CodeTrans: "SEBIS/code trans t5 small api generation multitask"
- CodeT5: "Salesforce/codet5-small"
- T5-NL: "t5-small"

commented-out code (based on the inclusion of keywords often found in commented-out code such as *printf()*) and comments that contain only whitespace or only contain special characters (used predominately to separate files into segments).

For input-output data samples, different pre-processing steps were applied to the input comment and output code part of the data, respectively. All single-line comments (for which the comment identifier is a semicolon and the comment extends until a newline character is encountered) were converted to multi-line comments (The body of the comment is surrounded by */* and */*). All comments in the output code were removed.

For MLM, the data was pre-processed by tokenizing, concatenating tokens from all SKILL files (separated by the *end of sequence* special token), and creating sequences of 512 tokens from the full sequence of tokens. These sub-sequences were the training samples given to the models. For each training sample, a masking percentage of 15% with a mean mask sequence length of 3 tokens was used, similar to how T5[12] pre-processed unlabeled data.

For Seq2Seq training and evaluation, a maximum input sequence length of 1024 and a maximum output sequence length of 512 were used. The input sequence length is longer because truncation of the input sequence is worse than truncation of the output sequence since the outputs are evaluated and are conditioned on the input sequences. This is especially the case if function definitions are truncated. To reduce the number of times input sequences have to be truncated, only the first 150 words (separated by whitespace) in comments of inputs are kept in the input sequence before truncation. This is similar to the strategy used for the CodeSearchNet dataset[38] where only the first paragraph in a documentation string is kept. Beam search with a beam size of 10 was used for all models during all evaluation procedures.

5. RESULTS & DISCUSSION

This section presents and discusses the experimental results of the methodology presented in Section 3. First, statistics of the custom SKILL dataset are shown in Section 5.1. Second, the validation results from training the pre-trained (or not in the case of T5-SKILL) models using different training strategies are shown in Section 5.2. This includes selecting the models for final evaluation by BLEU score. Finally, Section 5.3 presents the test results in terms of BLEU (using a SKILL-specific tokenizer), SKILL lint IQ, and human evaluation score.

5.1 SKILL Dataset Statistics

The custom SKILL dataset consists of file samples for self-supervised training and input-output pairs for supervised training and evaluation. The numbers of proprietary, open-source, and total SKILL files in the dataset are 983, 2265, and 3248, respectively. Table 2 shows the number of different SKILL files by source, split, and filtering strategy. The lint pass criterion is the most strict, filtering out more than half of all training files. Figure 6 shows a histogram for the SKILL lint results for each file. Filtering files using a minimum lint IQ score also removes many files. A minimum lint IQ score of 10 was chosen because a large number of files had a lower score (see Figure 6). Manually sampling and inspecting files with a lint score smaller than 10 confirmed that many are either very short or not high-quality SKILL files. Requiring the presence of input-output pairs leaves the largest number of remaining files. The non-filtered dataset was mined to create the supervised dataset as described in Section 3.1.3. Table 3 shows the number of input-output pairs by the dataset split.

Figure 7 shows a histogram of the number of input-output pairs mined for each file. Four hundred pairs were mined from one outlier file. Only 18 of these 400 pairs were top-level pairs meaning the other 382 pairs were removed after pair deduplication. This could be a reason why deduplication was found to be effective for training as shown in Subsection 5.2.

Table 4 compares the size of the custom SKILL dataset to the reported sizes of different subsets of the CodeTrans[27] training dataset. The number of files in the smallest file subset is almost 53 times larger than the number of files in the SKILL dataset. The number of snippet-level samples in the smallest subset by total samples is almost 10 times larger than the number of input-output pairs in the SKILL dataset. This shows that the custom SKILL dataset is extremely small compared to other code datasets. This is the main motivation for the use of weights pre-trained on general PL data and fine-tuning on both unlabeled and labeled data to use the small SKILL dataset as efficiently as possible. Still, transformers[9] tend to overfit small datasets. Increasing the size of the SKILL dataset will be needed in future work as discussed in Section 6.

Table 2. Number of files in the custom SKILL dataset by source and split. The number of training files that remain after three different filtering methods is also shown.

Split	Filtering	Proprietary		Open-source		All
		Primary	Secondary	Repository search	Code search	
Train	-	21	548	820	1445	2834
	≥ 1 pairs mined	0	319	699	885	1903
	lint IQ ≥ 10	20	223	654	934	1831
	lint pass	18	305	532	588	1443
Val	-	207	0	0	0	207
Test	-	207	0	0	0	207
All	-	435	548	820	1445	3248
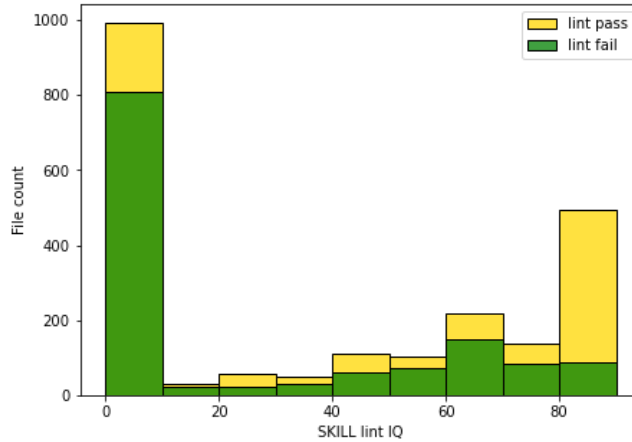
Figure 6. Stacked histogram of the SKILL lint IQ score of the training data files. Each bar is split by the number of files that received a "pass" or "fail" grade from the SKILL lint tool.

(a)   (b)
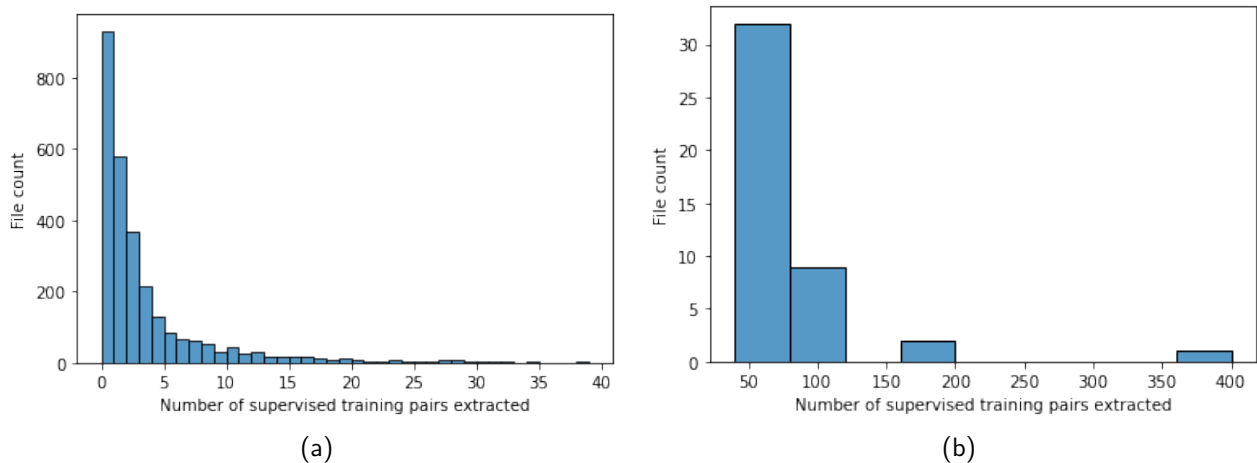
Figure 7. Histograms of training files by the number of supervised pairs extracted from them for 0-40 pairs (7a) and 40-400 pairs (7b). The histogram was split in two for visualization purposes.

5.2 Validation Results

The proposed models and the two baselines were trained and validated on the custom SKILL dataset. For each model type (CodeTrans,[27] CodeT5,[28] T5-NL,[12] and T5-SKILL), all training strategies that can be made through all possible combinations of (un-)supervised learning and the filtering and deduplication strategies described in Section 3.1.4 were applied to train a model.

Table 3. Number of comment-function (CF), comment-code (CC), and function-completion (FC) pairs in the custom SKILL dataset by the split.

Split	CF	CC	FC	All
Train (All)	1,212	6,766	4,564	12,542
Train (Deduplicated)	1,200	2,303	2,495	5,998
Validation	180	180	180	540
Test	177	177	177	531

Table 4. Size of the obtained SKILL training dataset compared with subsets of the dataset used to train CodeTrans. The subsets chosen are the two largest (Java and Python) and the four smallest (CSharp, Ruby, SQL, LISP) by the total number of samples categorized by PL.

Language	File-level samples	Snippet-level samples	Sources
Java	720,124	9,581,115	CodeSearchNet,[38] CodeNN,[36] DeepCom,[40] DeepAPI,[50] Public Git Archive[37]
Python	149,114	1,296,064	CodeSearchNet,[38] CodeNN,[36] Python150K[51]
CSharp	469,038	52,943	CodeNN,[36] Public Git Archive[37]
Ruby	0	179,281	CodeSearchNet[38]
SQL	0	158,862	CodeNN,[36] StaQC[52]
LISP	0	122,602	GitHub[35]
SKILL	2,834	12,542	Proprietary sources, GitHub[35]

Table 5 shows the results of the top three models in terms of BLEU score. Interestingly, both baselines achieved better BLEU scores without supervised training. It is believed that these results come from degenerated n-gram repeating and repeating tokens from the input which result in deceptively high BLEU scores. The best training strategy for each of these models which were trained in a supervised manner are also shown in Table 5 and these models were selected for final evaluation on the test set for further investigation. Another training strategy that was not the best of its model type but was chosen for evaluation on the test set was the CodeTrans model which was not fine-tuned in a self-supervised manner. This model was the third best CodeTrans model trained based on validation BLEU score. This was surprising since all other models pre-trained on general PL data do seem to benefit from self-supervised fine-tuning, based on the BLEU validation results. To investigate further, this model was selected for final evaluation on the test set.

Figure 8 shows an example output for each model selected for final evaluation. Ideally, these models were to output SKILL code that is functionally equivalent to the output reference. These outputs demonstrate that the models are not yet capable enough to output compilable SKILL programs. Reasons for this and how SKILL code generation performance can be improved are discussed in Section 6. The output of the best CodeTrans model is an example of repeating tokens, namely the "width" and "layer" tokens. The output of the best T5-NL model is an example of repeating tokens from the input. The differences in the model outputs despite all the models being given the same input show that the pre-training data, fine-tuning strategy, and tokenizers affect their generation capabilities.

All the best models that were trained in a supervised manner were trained on the deduplicated supervised training set. The top models trained in a supervised manner for every model type used file-filtering for self-supervised training with the results suggesting that there was a correlation between the size of the pre-training code dataset and the amount of unlabeled data that is filtered. For example, the top CodeTrans[27] model used the largest code dataset for pre-training and uses the filtering technique that filters the most number of SKILL files, the SKILL lint pass grade. Whether including comments in the self-supervised training set helps the models learn is inconclusive from the results obtained.

Figure 9 shows the training and validation loss curves for the models selected for final evaluation. The validation loss curves during self-supervised learning (Figure 9a) of pre-trained models start to saturate within the first 7 epochs of training. The models trained from scratch do not saturate even after 50 epochs and have higher validation losses than the pre-trained models. This suggests that pre-training helps the model learn the

Table 5. Evaluation results for the top three models, and interesting outliers, in terms of BLEU score on the validation dataset split. Here we denote a model as a model type trained using a certain training strategy (columns 3-7). The rank of a model in terms of BLEU score (final column) relative to other models of the same model type is shown in the second column. Models that were selected for final evaluation on the test set have * next to their rank.

Model type	Rank	Self-supervised training			Supervised		BLEU
		applied?	file filtering?	comments?	applied?	deduplicated?	
CodeTrans	1*	✓	lint pass	✓	✓	✓	0.0864
	2	✓	≥ 1 pairs mined	✓	✓	✓	0.0739
	3*	✗	NA	NA	✓	✓	0.0676
CodeT5	1*	✓	lint IQ ≥ 10	✗	✓	✓	0.0470
	2	✓	-	✗	✓	✓	0.0465
	3	✓	-	✓	✓	✓	0.0455
T5-NL	1*	✓	-	✗	✗	NA	0.0623
	2	✓	lint pass	✓	✗	NA	0.0619
	3	✓	lint pass	✗	✗	NA	0.0567
	4*	✓	≥ 1 pairs mined	✓	✓	✓	0.0522
T5-SKILL	1*	✓	-	✗	✗	NA	0.0829
	2	✓	lint IQ ≥ 10	✗	✗	NA	0.0820
	3	✓	lint IQ ≥ 10	✓	✗	NA	0.0737
	9*	✓	≥ 1 pairs mined	✓	✓	✓	0.0376

SKILL language faster and better. The validation loss curves for supervised learning (Figure 9b) do not improve after the first epoch. This suggests that the supervised learning task is not as effective for learning to model the SKILL language as it quickly starts to overfit. However, as the next section will show, supervised training is still beneficial in terms of BLEU score and human judgment score for models pre-trained on general PL data.

5.3 Final Evaluation Results

The selected models from Table 5 were evaluated on the test split of input-output pairs. Figure 10 shows the scores assigned to each model for each sample in the human evaluation study and the mean scores for each pair type. Function-completion pairs were given a score of 1 for every sample. The survey results gave mean scores of 4.4 and 4.2 out of 5 for the quality of the input prompt and output references, respectively, for the function-completion samples so the comprehensiveness of the function-completion pairs is most likely not the problem. The results for comment-function and comment-code pairs were better for most of the models. Both T5-SKILL models performed poorly which supports the hypothesis that the custom SKILL dataset is too small to train a model from scratch. These results also suggest that the reason for the poor results on the function-completion pairs might be because the pre-trained models have been trained on a lot of NL and so this knowledge has transferred much better than the models have been able to learn the SKILL language.

The CodeT5 model[28] achieved the highest overall mean score of 2.27. The CodeTrans model[27] achieved the next-best mean score of 1.93. The T5-NL model[12] with supervised training achieved a mean score of 1.67, a better score than the T5-NL model not trained in a supervised manner and the CodeTrans model without self-supervised training. This suggests that both self-supervised and supervised training are beneficial. When asked to score whether the best outputs for each sample would be useful if the evaluator was tasked with writing SKILL code for the corresponding input prompts, the human evaluator gave scores of 1, 2, and 1 out of 5 for comment-function, comment-code, and function-completion pairs, respectively. This refutes the idea that the models trained on the custom SKILL dataset are ready to be deployed in a real-world setting to assist SKILL developers.

The two automatic evaluation metrics, BLEU using different tokenizers and lint IQ, were measured on the same samples from the human evaluation survey to calculate the correlation between them. Figure 11 shows

Input
```
/* Create a Pcell that generates rectangles along the x axis which fit within
a bounding box with width 15. This pcell accepts the following parameters:
width    X-dimension of the rectangle. (float, default = 1.0)
length   Y-dimension of the rectangle. (float, default = 5.0)
cd1      Space between rectangles in the X-dimension (float, default = 1.5)
layer    layer type of the rectangles (string, default = "metal1")
*/
pcDefinePCell(
    list(ddGetObj("pcells") "rectanglesRow" "layout")
    ( (width 1.0) (length 5.0)
      (cd1 1.5)  (layer "metal1") )
```

Output reference
```
let((tfId bbox) bbox = list(0:0 15:length+1) masterRect = rodCreateRect(
?cvId pcCellView ?layer list(layer "drawing") ?width width
?length length ?spaceX cd1 ?fillBBox bbox) ) )
```

CodeTrans
```
(create_padtileArray (width width width width width width height width width
width width layer layer layer layer layer "metal1" "layout" "maskLayout"
"fixed" "R0" "fixed" nil "fixed" nil nil "fixed" ) ) )
```

CodeT5
```
   (width 1.0    (length 5.0)    (length 3.0)    ) )
```

T5-NL
```
(createMosaic_rectanglesRow (width width) (width width) ) )
```

T5-SKILL
```
" "layout") ( (width 1.0) (length 5.0) ) ) layer layer type of the rectangles
( string, default = " metal1 ")*/ pcDefinePCell( list(ddGetObj("pcells")
"rectangles) ( (length 1.0) (width 5.0) (cd1 1.5) (layer "metal1") )  y axis
```

CodeTrans Sup-Only
```
type : width y  length rectangles in the x width  - width y =  box with size
layer   pcdefinepcell( list(ddgetobj("lib_webers_generic") )
```

T5-NL+Sup
```
pcDefinePCell(ddGetObj("pcells") "rectanglesRow" "layout") (width 1.0)
(length 5.0) (cd1 1.5) (layer "metal1") ) )
```

T5-SKILL+Sup
```
layer = "metal1"
```

Input-output pair

Model output predictions

Figure 8. Model predictions for the example input prompt of the program on the left of Figure 3. Note that the output reference has been formatted to save space. Newlines were manually added to the model outputs to save space and improve readability.

that the correlation between the change in the lint IQ score correlates well with human scores compared to BLEU scores. Since the function-completion human-assigned scores were all the same, these results are counter-productive for the calculation of the correlations and have not been included in the calculations of the coefficients. The high correlation of the changes in SKILL lint IQ with human evaluation scores might be attributable to the overall low quality of code synthesized by the models. Synthesized code that was syntactically and semantically correct was probably judged relatively favorably by human evaluators since little to none of the generated code followed logically from the input prompt. That is, the functional correctness of the SKILL code generated from all the models remained consistently low.

For BLEU scores, the best correlations are achieved when the monotonic function-completion scores of the human evaluation are ignored. As the models improve, especially by reducing the frequency of degenerate token copying and repetition (feedback received from the open-ended questions on the human judgment survey), BLEU scores should correlate better with human evaluation scores. The tokenizer that achieved the best correlation between BLEU and human scores is the T5-SKILL tokenizer. This was to be expected since this tokenizer was trained on the custom SKILL dataset and therefore has a vocabulary that is most relevant for SKILL. Therefore, the T5-SKILL tokenizer was used for the standardized BLEU score. Surprisingly, the T5-NL tokenizer[12]
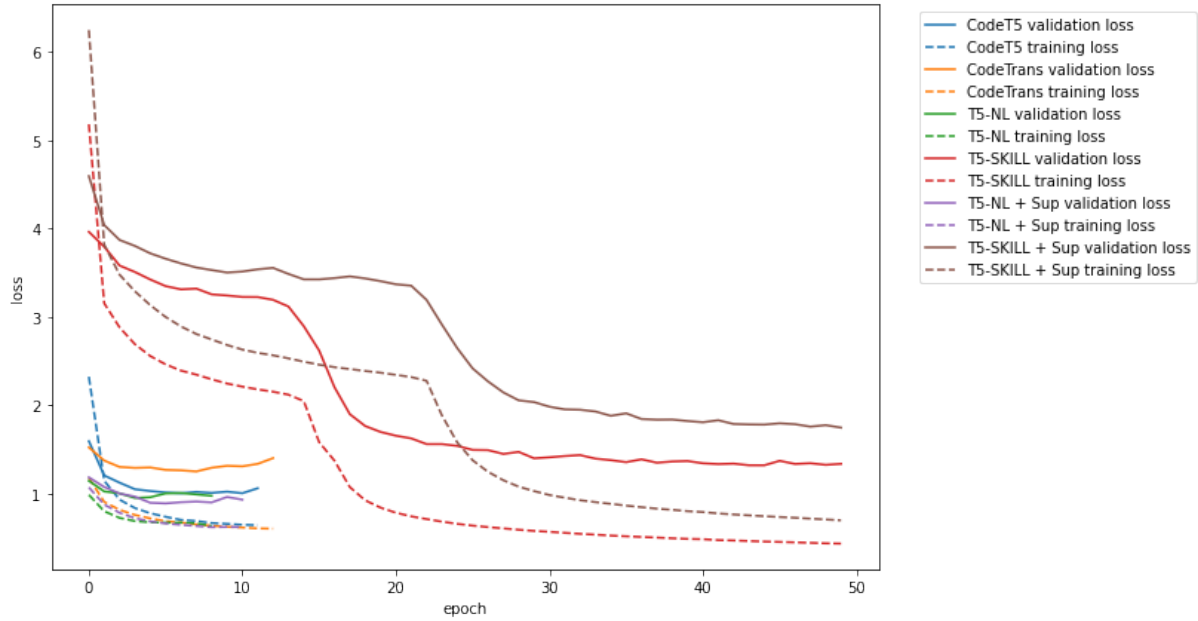
(a)
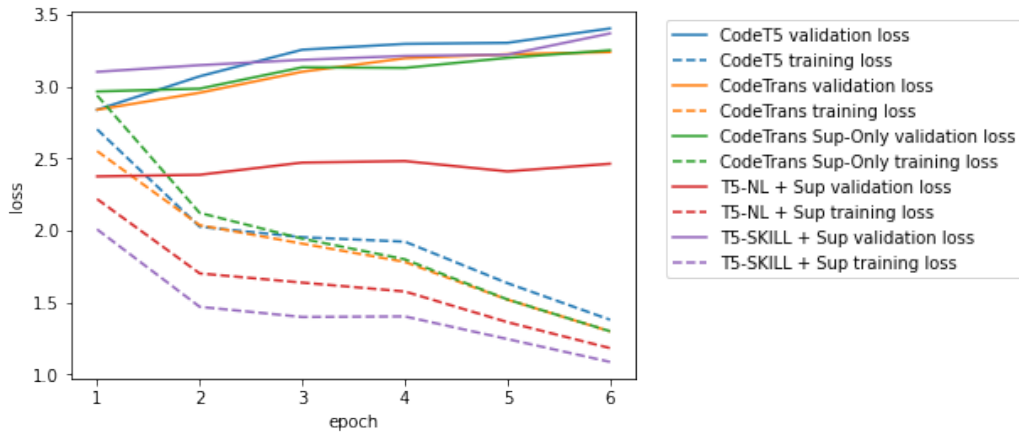
(b)

Figure 9. Self-supervised (9a) and supervised (9b) training and validation loss curves for models selected for final evaluation.

correlated better with human scores than the tokenizers of the proposed models which suggests that general PL tokenizers might not be very relevant for the SKILL PL.

The final results on the entire test set (except for human evaluation scores) for the selected models are shown in Table 6. The CodeTrans model[27] that achieved the best validation BLEU score also achieved the best SKILL standardized BLEU score of 0.032. This is double or more than what all non-CodeTrans models achieve. CodeTrans (with and without self-supervised training), CodeT5,[28] and T5-NL[12] with supervised training achieved similar mean changes in lint IQ scores. As discussed above, CodeT5 performed best in the human evaluation with CodeTrans performing second best. These results support the hypothesis that models pre-trained on large volumes of PL data can more efficiently learn unseen PLs than models pre-trained only on NL or models trained from random initial weights. The results also support the hypothesis that fine-tuning using both supervised and self-supervised learning improves SKILL code generation models.
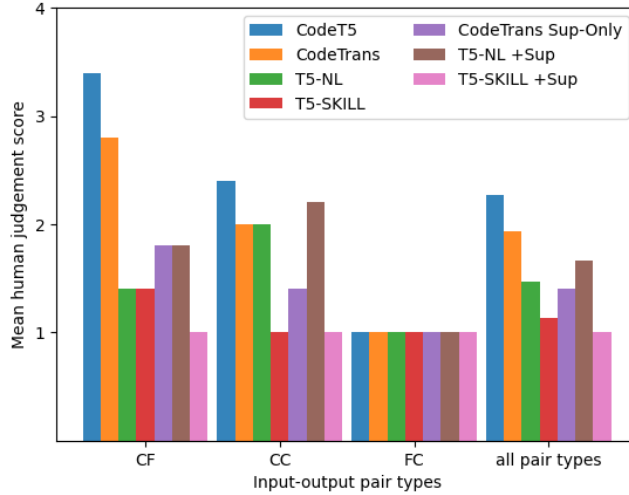
Figure 10. Mean human evaluation scores for the different types of pairs as well as the overall mean score for all pair types. CF, CC, and FC stand for comment-function, comment-code, and function-completion pairs, respectively.

Figure 11. Pearson correlation coefficients of BLEU variants using different tokenizers (corresponding to the four model types) and the change in lint IQ score with comment-function and comment-code human evaluation scores. The BLEU-n scores are the BLEU scores when only considering n-gram precision. "BLEU-Mean" denotes the same BLEU score used in the rest of the paper (geometric mean of the 1,2,3,4-gram precisions).

However, the overall results of the models indicate that the models are not yet capable of reliable SKILL code generation. If the models were generating quality SKILL code, we could expect the mean change in lint IQ to be in the range of -30 and -10, indicating mostly style and efficiency errors instead of significant syntax and compilability errors. Furthermore, the mean human scores should be at least 5 to indicate that most generated code snippets are at least plausible to be correct SKILL statements to a SKILL developer. The next section discusses the most likely reasons for these poor results. The BLEU score results are harder to interpret without relating to baselines. Subsection 6.3 discusses BLEU's viability as a SKILL code generation metric for future works.

Table 6. Final evaluation results on the input-output pair test split of the custom SKILL dataset for different models in terms of BLEU, T5-SKILL-BLEU (BLEU score calculated using the T5-SKILL tokenizer), and change in SKILL lint IQ metric. The mean human evaluation scores for each model for 15 pairs (see Section 3.3) are provided in the final column.

Model	BLEU	T5-SKILL-BLEU	Mean Δ Lint IQ	Mean Human Score
CodeTrans	**0.019**	**0.032**	**-60.7**	1.93
CodeT5	0.008	0.015	**-60.3**	**2.27**
T5-NL	0.004	0.007	-70.0	1.46
T5-SKILL	0.016	0.016	-71.7	1.13
CodeTrans Sup-Only	0.014	0.024	**-59.4**	1.40
T5-NL +Sup	0.002	0.002	**-59.8**	1.67
T5-SKILL +Sup	0.002	0.002	-64.4	1.00

6. LIMITATIONS & PROPOSED FUTURE WORK

The results of the experiments discussed in the previous section suggest that more work must be done before the proposed methodology can be deployed to assist SKILL developers. In this section, the main limitations of

the proposed methodology are discussed. These limitations are dataset scaling and confidentiality, size of the models, and evaluating SKILL code. For each limitation, promising research directions for future work that address these limitations are proposed.

6.1 Dataset Scaling & Confidentiality

Table 4 showed that the size of the custom SKILL dataset is orders of magnitude smaller than datasets of other PLs. The amount of open-source SKILL data is small relative to general PLs such as Python[34] or Java[53] largely because the SKILL language itself is proprietary. While there are a large number of proprietary SKILL repositories, a very limited number of them were able to be used for the experiments in this study due to confidentiality concerns. Future work in privacy-preserving ML, such as homomorphic encryption[54] or federated learning,[55] could alleviate these confidentiality concerns and allow for data collection across teams and organizations.

GitHub data was not checked for permissive licenses. For this study, this should not be a problem since the code was only forked and not further distributed nor was it profited from. Furthermore, O'Keefe et al.[56] argued that training ML models on copyrighted works should constitute fair use but there is still legal ambiguity. Even if there are no legal issues, the use of free and open-source software to create proprietary products presents an ethical concern for the developers of such software.[57]

6.2 Model Size

All the models trained in this study are based on the "small" T5 architecture.[12] Chowdhery et al.[58] showed that increasing the size of a language model not only generally improves performance but that certain tasks require models with a certain minimum number of parameters to obtain acceptable results. The relationship between the inputs and outputs of the supervised dataset is complex and a higher number of parameters might be necessary to model it effectively. Small architectures were used due to a combination of limited computation resources and a large number of different models trained (72).

Future work could use the results from this study to limit the model search space allowing for fewer but larger models to be explored. However, larger models require more computing resources which can lead to higher energy consumption and in turn leads to high financial costs and negative climate change effects.[59] Additionally, high computing requirements usually lead to the centralization of the model on a server where client nodes can send requests too. This increases the risk of breaches in data privacy which is important to the semiconductor industry as discussed in the previous subsection.

6.3 Evaluating SKILL Code

Despite its popularity, Ren et al.[60] argue against using metrics intended for the evaluation of NL (such as BLEU) for evaluating code. This is because functionally equivalent programs can consist of different tokens and BLEU does not take this into account. Instead, Ren et al.[60] proposes CodeBLEU which is inspired by the original BLEU metric[47] which is more appropriate for evaluating synthesized code. CodeBLEU uses a weighted sum of BLEU, BLEU weighted on code keywords, syntactic similarity by comparing ASTs, and data-flow similarity. CodeBLEU could not be used for the SKILL code generation experiments since the SKILL parser is not publicly available and so there was no simple way to obtain ASTs of SKILL code. Since the correlation between changes in SKILL lint IQ scores and human judgments was shown to be relatively strong in Section 5.3, future work should investigate how well a combination of BLEU and SKILL lint IQ correlate with human judgments.

SKILL lint IQ is an automatic metric that requires no reference code. A promising direction for future work would be to sample multiple outputs from the models and use the lint IQ score to choose the best output as the final prediction. Multiple outputs can be sampled from language models through generation strategies such as top-k sampling.[61] This is inspired by, but different from, related works that achieve improved functional correctness metrics by sampling a large number of outputs and relying on unit tests to filter candidate predictions.[10, 23]

When a model is developed that can reliably generate SKILL code with high SKILL lint IQs, the functional correctness of the generated code can be evaluated. This requires verification programs for each test program. For physical layout generation, this will require PCells to be paired with a list of parameter values and corresponding

physical layout instantiations. Preferably, programs to measure functional correctness should be mined from real-world data instead of manually writing SKILL code for functional correctness. This is discussed in more detail in Section 3.3.

The sample size of the human evaluation study was small, with 15 different prompts for each model. This means that a high variance of the human evaluation results can be expected were the study to be replicated using different prompts or a different human evaluator. Future work should ensure that human evaluations of SKILL code are easier or more accessible to allow for more samples to be evaluated.

7. CONCLUSION

Token-based code generation using deep learning transformer models promises to improve the productivity of programmers. These transformer models generally require large code datasets to learn to model a programming language effectively. In this work, we present the first study on transformer-based SKILL code autocompletion towards improving the productivity of design engineers facing increasingly more complex hardware design challenges. We propose a novel, data-efficient methodology for training models to autocomplete SKILL code. This methodology includes the creation of a high-quality SKILL dataset that contains unlabeled and labeled data. It also includes fine-tuning T5-based models pre-trained on general PL data on both the unlabeled and labeled data.

Validation results suggested that applying a file quality filter to the self-supervised training set and deduplicating pairs in the supervised training set leads to improved BLEU scores. A human evaluation study was conducted and showed that BLEU scores obtained using a standard, SKILL-specific tokenizer correlated better to human judgments than using other tokenizers. It also showed that changes in SKILL lint IQ, a static analysis score based on syntactic and semantic correctness, correlated relatively well with human judgments. Test results showed that using models pre-trained on general PL data and fine-tuned using both unlabeled and labeled SKILL data improved BLEU, SKILL lint IQ, and human evaluation scores. However, the test results refute the viability of the trained models for reliable SKILL code autocompletion. Limitations of the study and corresponding suggestions for future work to address these limitations are discussed. Ultimately, the experimental results obtained provide valuable insights towards autocompleting SKILL code in a data-efficient manner.

Acknowledgments

REFERENCES

[1] Maxfield, C. M., "Chapter 17 - application-specific integrated circuits (asics)," in [*Bebop to the Boolean Boogie (Third Edition)*], Maxfield, C. M., ed., 235–249, Newnes, Boston, third edition ed. (2009).

[2] Intel, "Moore's law and intel innovation." accessed: 2022-05-23.

[3] Carver, M. and Lynn, C., "Introduction to vlsi systems," *Reading, MA, Addison-Wesley Publishing Co., 1980. 426 p.* **-1** (01 1980).

[4] Cadence Design Systems, "Computational software for intelligent system design." accessed: 2022-05-25.

[5] Barnes, T., "Skill: a cad system extension language," in [*27th ACM/IEEE Design Automation Conference*], 266–271 (1990).

[6] Cadence Design Systems, "Virtuoso layout suite." accessed: 2022-05-23.

[7] Cadence Design Systems, "Allegro pcb designer." accessed: 2022-05-23.

[8] Huang, G., Hu, J., He, Y., Liu, J., Ma, M., Shen, Z., Wu, J., Xu, Y., Zhang, H., Zhong, K., Ning, X., Ma, Y., Yang, H., Yu, B., Yang, H., and Wang, Y., "Machine learning for electronic design automation: A survey," (2021).

[9] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I., "Attention is all you need," *CoRR* **abs/1706.03762** (2017).

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

[11] Github, "Github copilot - your ai pair programmer." accessed: 2022-03-06.

[12] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J., "Exploring the limits of transfer learning with a unified text-to-text transformer," *CoRR* **abs/1910.10683** (2019).

[13] Tayenjam, S., Vanukuru, V. N. R., and Kumaravel, S., "A pcell design methodology for automatic layout generation of spiral inductor using skill script," in [*2017 International conference on Microelectronic Devices, Circuits and Systems (ICMDCS)*], 1–4 (2017).

[14] Abhishek, K., Harini, K., Rachana, B., Sobhana, T., and Dhanabal, R., "Design and analysis of on-chip spiral inductors using automatic generated layouts through skill code," in [*2018 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*], 887–891 (2018).

[15] Zeng, H., Zhang, C., and Prasanna, V., "Fast generation of high throughput customized deep learning accelerators on fpgas," in [*2017 International Conference on ReConFigurable Computing and FPGAs (Re-ConFig)*], 1–8 (2017).

[16] Esmaeilzadeh, H., Ghodrati, S., Gu, J., Guo, S., Kahng, A. B., Kim, J. K., Kinzer, S., Mahapatra, R., Manasi, S. D., Mascarenhas, E., Sapatnekar, S. S., Varadarajan, R., Wang, Z., Xu, H., Yatham, B. R., and Zeng, Z., "Verigood-ml: An open-source flow for automated ml hardware synthesis," in [*2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*], 1–7 (2021).

[17] Moreira, T. G., Wehrmeister, M. A., Pereira, C. E., Pétin, J.-F., and Levrat, E., "Automatic code generation for embedded systems: From uml specifications to vhdl code," in [*2010 8th IEEE International Conference on Industrial Informatics*], 1085–1090 (2010).

[18] Madorsky, A. and Acosta, D. E., "Vpp - a verilog hdl simulation and generation library for c++," in [*2007 IEEE Nuclear Science Symposium Conference Record*], **3**, 1927–1933 (2007).

[19] Takamaeda-Yamazaki, S., "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in [*Applied Reconfigurable Computing*], Sano, K., Soudris, D., Hübner, M., and Diniz, P. C., eds., 451–460, Springer International Publishing, Cham (2015).

[20] Yankova, Y., Kuzmanov, G., Bertels, K., Gaydadjiev, G., Lu, Y., and Vassiliadis, S., "Dwarv: Delftwork-bench automated reconfigurable vhdl generator," in [*2007 International Conference on Field Programmable Logic and Applications*], 697–701 (2007).

[21] Watson, C., Tufano, M., Moran, K., Bavota, G., and Poshyvanyk, D., "On learning meaningful assert statements for unit test cases," in [*2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*], 1398–1409 (2020).

[22] Korbak, T., Elsahar, H., Kruszewski, G., and Dymetman, M., "Controlling conditional language models with distributional policy gradients," *CoRR* **abs/2112.00791** (2021).

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

[24] Hong, J., Dohan, D., Singh, R., Sutton, C., and Zaheer, M., "Latent programmer: Discrete latent codes for program synthesis," (2020).

[25] Dehaerne, E., Dey, B., Halder, S., De Gendt, S., and Meert, W., "Code generation using machine learning: A systematic review," *IEEE Access* **10**, 82434–82455 (2022).

[26] Hassan, M., Mahmoud, O., Mohammed, O., Baraka, A., Mahmoud, A., and Hassan Yousef, A., "Neural machine based mobile applications code translation," 302–307 (10 2020).

[27] Elnaggar, A., Ding, W., Jones, L., Gibbs, T., Feher, T., Angerer, C., Severini, S., Matthes, F., and Rost, B., "Codetrans: Towards cracking the language of silicone's code through self-supervised deep learning and high performance computing," *CoRR* **abs/2104.02443** (2021).

[28] Wang, Y., Wang, W., Joty, S., and Hoi, S. C. H., "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," (2021).

[29] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C., "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint* (2022).

[30] Pearce, H., Tan, B., and Karri, R., "Dave: Deriving automatically verilog from english," in [*2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*], 27–32 (2020).

[31] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I., "Language models are unsupervised multitask learners," (2019).

[32] Thakur, S., Ahmad, B., Fan, Z., Pearce, H., Tan, B., Karri, R., Dolan-Gavitt, B., and Garg, S., "Benchmarking large language models for automated verilog rtl code generation," in [*2023 Design, Automation Test in Europe Conference Exhibition (DATE)*], 1–6 (2023).

[33] The Selenium project, "Webdriver." accessed: 2022-08-12.

[34] Python Software Foundation, "Welcome to python.org." accessed: 2022-03-06.

[35] Github, "Github: Where the world builds software." accessed: 2022-03-06.

[36] Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L., "Summarizing source code using a neural attention model," in [*Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*], 2073–2083, Association for Computational Linguistics, Berlin, Germany (Aug. 2016).

[37] Markovtsev, V. and Long, W., "Public git archive: a big code dataset for all," *CoRR* **abs/1803.10144** (2018).

[38] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M., "Codesearchnet challenge: Evaluating the state of semantic code search," (2019).

[39] Microsoft, "Intermediate language & execution." accessed: 2022-05-08.

[40] Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z., "Deep code comment generation," in [*Proceedings of the 26th Conference on Program Comprehension*], *ICPC '18*, 200–210, Association for Computing Machinery, New York, NY, USA (2018).

[41] LeClair, A. and McMillan, C., "Recommendations for datasets for source code summarization," (2019).

[42] Barone, A. V. M. and Sennrich, R., "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *CoRR* **abs/1707.02275** (2017).

[43] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M., "Transformers: State-of-the-art natural language processing," in [*Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*], 38–45, Association for Computational Linguistics, Online (Oct. 2020).

[44] Kudo, T. and Richardson, J., "SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in [*Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*], 66–71, Association for Computational Linguistics, Brussels, Belgium (Nov. 2018).

[45] Sennrich, R., Haddow, B., and Birch, A., "Neural machine translation of rare words with subword units," in [*Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*], 1715–1725, Association for Computational Linguistics, Berlin, Germany (Aug. 2016).

[46] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D., "Language models are few-shot learners," (2020).

[47] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J., "Bleu: a method for automatic evaluation of machine translation," in [*Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*], 311–318, Association for Computational Linguistics, Philadelphia, Pennsylvania, USA (July 2002).

[48] Post, M., "A call for clarity in reporting BLEU scores," in [*Proceedings of the Third Conference on Machine Translation: Research Papers*], 186–191, Association for Computational Linguistics, Brussels, Belgium (Oct. 2018).

[49] Shazeer, N. and Stern, M., "Adafactor: Adaptive learning rates with sublinear memory cost," (2018).

[50] Gu, X., Zhang, H., Zhang, D., and Kim, S., "Deep API learning," *CoRR* **abs/1605.08535** (2016).

[51] Bielik, P., Raychev, V., and Vechev, M., "Phog: Probabilistic model for code," in [*Proceedings of The 33rd International Conference on Machine Learning*], Balcan, M. F. and Weinberger, K. Q., eds., *Proceedings of Machine Learning Research* **48**, 2933–2942, PMLR, New York, New York, USA (20–22 Jun 2016).

[52] Yao, Z., Weld, D. S., Chen, W., and Sun, H., "Staqc: A systematically mined question-code dataset from stack overflow," *CoRR* **abs/1803.09371** (2018).

[53] Oracle, "Java." accessed: 2022-03-06.

[54] Pulido-Gaytan, L. B., Tchernykh, A., Cortés-Mendoza, J. M., Babenko, M., and Radchenko, G., "A survey on privacy-preserving machine learning with fully homomorphic encryption," in [*High Performance Computing*], Nesmachnow, S., Castro, H., and Tchernykh, A., eds., 115–129, Springer International Publishing, Cham (2021).

[55] Zhang, C., Xie, Y., Bai, H., Yu, B., Li, W., and Gao, Y., "A survey on federated learning," *Knowledge-Based Systems* **216**, 106775 (2021).

[56] O'Keefe, C., Lansky, D., and Clark, J., "Comment regarding request for comments on intellectual property protection for artificial intelligence innovation." accessed: 2022-14-08.

[57] Gingerich, D. and Kuhn, B. M., "Give up github: The time has come!." accessed: 2022-08-14.

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

[59] Bender, E. M., Gebru, T., McMillan-Major, A., and Shmitchell, S., "On the dangers of stochastic parrots: Can language models be too big?," in [*Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*], FAccT '21, 610–623, Association for Computing Machinery, New York, NY, USA (2021).

[60] Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S., "Codebleu: a method for automatic evaluation of code synthesis," (2020).

[61] Fan, A., Lewis, M., and Dauphin, Y., "Hierarchical neural story generation," (2018).