NL2CMD: An Updated Workflow for Natural Language to Bash Commands Translation

Quchen Fu*, Zhongwei Teng, Marco Georgaklis,
Jules White and Douglas C. Schmidt

Dept. of Computer Science, Vanderbilt University Nashville, TN, USA
E-mail: quchen.fu@vanderbilt.edu; zhongwei.teng@vanderbilt.edu;
marco.georgaklis@vanderbilt.edu; jules.white@vanderbilt.edu;
d.schmidt@vanderbilt.edu
**Corresponding Author*

Received 09 July 2022; Accepted 16 September 2022;
Publication 31 January 2023

Abstract

Translating natural language into Bash Commands is an emerging research field that has gained attention in recent years. Most efforts have focused on producing more accurate translation models. To the best of our knowledge, only two datasets are available, with one based on the other. Both datasets involve scraping through known data sources (through platforms like stack overflow, crowdsourcing, etc.) and hiring experts to validate and correct either the English text or Bash Commands.

This paper provides two contributions to research on synthesizing Bash Commands from scratch. First, we describe a state-of-the-art translation model used to generate Bash Commands from the corresponding English text. Second, we introduce a new NL2CMD dataset that is automatically generated, involves minimal human intervention, and is over six times larger than prior datasets. Since the generation pipeline does not rely on existing Bash Commands, the distribution and types of commands can be custom

Journal of Machine Learning Theory, Applications and Practice, Vol. 1, 45–82.
doi: 10.13052/jmltapissn.2023.002

adjusted. Our empirical results show how the scale and diversity of our dataset can offer unique opportunities for semantic parsing researchers.

Keywords: Bash Commands generation, NL2CMD dataset, semantic parsing, natural language processing.

1 Introduction

Automating the conversion of natural language to executable computer programs is a long-coveted goal that has recently experienced a resurgence of interest amongst researchers and practitioners. In particular, converting natural language to Bash (which is a shell scripting language for UNIX systems) has emerged as an area of interest, with the goal of automating repetitive tasks, such as file manipulation, search, and application-specific scripting.

The NL2Bash problem can be described as a semantic parsing challenge, i.e., creating a mapping from natural language to a formal, executable representation [1]. Significant efforts to tackle this problem have been sparked by the NLC2CMD competition at the NeurIPS 2020 conference. Our recent work in this competition yielded an architecture that improves the state-of-the-art performance in translating natural language to Bash Commands from 13.8% to 53.2% [2]. The transformer model that we created for the NLC2CMD competition is currently the best-performing architecture for this problem [3].

Until recently, the Bash translation problem has relied heavily on the availability of NL2Bash dataset [4]. This corpus of over 9,000 English-command pairs contains frequently used Bash Commands scraped from forums, tutorials, tech blogs, and course materials. Constructing the NL2Bash corpus involved hiring freelance software engineers, each assigned to manually search, browse, and enter data through a web interface.

The freelancers working on this project constructed roughly 50 english language and Bash Command pairs per hour, prior to filtering and cleaning the dataset [4]. This manual approach is clearly resource-intensive since it requires labor from specialized freelancers, which is time-consuming and expensive. Moreover, the approach is not scalable since the marginal cost of labor does not significantly diminish as the size of the dataset increases.

As commonly observed in this research domain, only marginal improvements in the accuracy of solutions to the NL2Bash problem have occurred since progress has been impeded due to the limited amount of annotated

data. This paper extends our prior published work [5] on translating natural language to Bash Commands and provides the following new contributions beyond our prior work:

1. It describes the use of a membership query synthesis technique to generate a large dataset of Bash Commands, expanding the available data to solve this problem,
2. It demonstrates a back-translation technique that takes the generated Bash Commands and creates corresponding natural language pairs,
3. It discusses a validation and verification technique for generated Bash Commands by converting them into an executable form and running them in an isolated environment that doubles as a data quality metric,
4. It presents a new dataset called NLC2CMD, which is the largest dataset for translating natural language to Bash Command available to researchers and practitioners,
5. It adopts a post-process addition to the original workflow, replacing placeholder values with actual arguments and making many of the translated commands executable in the Linux environment.

Most importantly, our work suggests that the future of tackling this hard problem lies within the automation of Bash Command Synthesis and Back-translation to natural language representations. We have established a workflow that can be improved upon and used to maximize model performance with minimal labeling costs. Our approach has numerous advantages over prior work, with notable improvements in time efficiency, labeling costs, diversity of the dataset, and practicality.

The remainder of this paper is organized as follows: Section 2 and 3 introduces the NLC2CMD problem and outlines recent developments in semantic parsing; Section 4 summarizes the challenges for translating natural language to Bash Commands; Section 5 analyzes the performance of different model structures and training techniques; Section 6 describes our data generation/validation technique and statistics including data quality and comparison with existing datasets; Section 7 discusses different metrics and error analysis for the state-of-the-art model on our new dataset; and Section 8 presents concluding remarks and outlines our future work.

2 Research Problem Overview

Translating natural language into source code for software or scripts can help developers find ways to accomplish tasks in languages they are not familiar

with, similar to how help forums like Stack Overflow are used today. As early as 1966, Sammet [6] envisioned a future of automated code generation where people program in their native language. While generating software templates from configuration files is now common practice, research on translating natural language into code is still in a relatively early stage.

Past research mainly focused on scripting languages or small code snippets. Various datasets have been created to aid research on generating code from natural languages. Examples of such datasets include WikiSQL for SQL [7], CoNaLa for Python [8], and NL2Bash for Bash [9].

This paper focuses on the task of translating natural language into Commands in the Bash scripting language. Translating natural language into Bash Commands is an example of semantic parsing, which means natural language is translated into logical forms that can be executed [10]. For example, the phrase "how do I compress a directory into a bz2 file" can be translated to the Bash command: `tar -cjf FILE_NAME PATH`.

In the near term, natural language to Bash Commands translation is unlikely to replace discussion groups or help forums completely. They can, however, provide a quick reference mechanism that may improve on-demand code suggestions and popups generated by integrated development environments (IDEs). This type of AI-based approach complements other prior work, such as SOFix [11], which can fix bugs in code by mining postings in Stack Overflow.

3 Background and Related Work

This section summarizes the background and related work surrounding the areas covered in this paper. Our contributions focus on advancing machine translation, where our approach is based on dataset synthesis. This research approach is novel in the domain of Bash Command translation.

3.1 Machine Translation

Various architectures have been explored on different tasks of program synthesis from natural language. For example, Lin et al. [12] achieved state-of-the-art generation of shell scripts using Recurrent Neural Networks (RNNs) [13]. Likewise, Zeng et al. [14] utilized the Bert [15]-based encoder and a pointer-generator [16] decoder to generate SQL code from text. Moreover, ValueNet [17] (Transformer encoder + LSTM decoder with pointer networks [18]) was the first Text-to-SQL system incorporating values.

In addition, Xu et al. [19] improved upon the TranX [20] transition-based neural semantic parser to translate natural language into general programming languages, such as Python.

The best results in prior work on the problem of translating natural language to Bash Commands were produced by Tellina [12]. Tellina used the Gated Recurrent Unit (GRU) Network [21], which is an RNN that achieved 13.8% accuracy on the NLC2CMD metrics proposed by IBM [22]. The Tellina [12] paper produced the NL2Bash [9] dataset and new semantic parsing methods that set the baseline for mapping English sentences to Bash Commands.

Transformer models generally have better accuracy and faster training times [23] than RNNs [13] on machine translation tasks. Prior research on machine translation has largely focused on the GRU architecture to translate natural language to Bash Commands. This paper enhances prior research by exploring the performance of several architectures on the NLC2CMD dataset.

Our experiments with applying Transformer models to the natural language to Bash task show that they outperform other approaches, such as (1) RNNs that show an 18.4% improvement and (2) Bidirectional RNN (BRNN) that show up to 4.4% improvement [24]. Analyzing how model structural choices and prediction strategies affect model performance in natural language to command translation task [22] is thus a key contribution of this paper. Since the energy and accuracy metrics for model evaluation were specifically designed for the NL2CMD competition, potential improvements for the metrics are also discussed.

3.2 Dataset Synthesis

Bash is a frequently-used command line scripting language. It thus offers a unique opportunity to generate diverse – and more importantly – executable commands easily due to its relatively short and simple nature. To increase programmer productivity, the Bash Commands suggested by a tool should be both syntactically and semantically correct. If suggestions are not syntactically correct and cannot execute, programmers may simply ignore them since they distract from the task at hand. Moreover, if translations are not semantically correct, programmers may execute Bash Commands that do not accomplish the goal that they want to achieve, or worse, have negative impacts on the system (such as deleting important files or directories).

Our work on Bash Command generation divides the synthesis into two steps: (1) scraping syntax and flag structures from the Bash manual pages

for efficient command generation and (2) training a back-translation model for accurate command summarization. This approach enabled us to construct a dataset of English-command pairs that is over six times larger than the original NL2Bash dataset.

Bash manual pages give an introduction to Bash features and are *the definitive reference on shell behavior* [25], providing complete and accurate guidance for Bash usage. Recent work has explored the use of manual page data for assistance in Bash to natural language translation [26], processing the page descriptions to aid the translation model. However, we found the manual pages offered additional insight and enough context into utility-flag relationships to generate an entirely new dataset from scratch.

Numerous approaches have incorporated dataset synthesis and augmentation in translation tasks. Nguyen et al. [27] explored the use of combining augmented data with the original dataset to boost the accuracy of neural machine translation between human languages. Zhao et al. [28] also explored data augmentation in neural machine translation to improve dataset diversification. Notably, Agarwal et al. [29] proposed using document similarity methods to create noisy parallel datasets of code, enabling the advancement of machine translation with monolingual datasets.

With dataset generation, transformer-based models have proven effective for parallel corpus mining in the domain of machine translation [30]. Previous research has tried using classification techniques, such as document similarity [29], to identify translations from pre-existing corpora.

4 Key Research Challenges

This section summarizes key research challenges we encountered when translating natural language to Bash Commands and describes general obstacles the machine translation community is facing on these topics.

4.1 Challenge 1: Translating From an Ambiguous Language to Precise Bash Commands is Hard

Translating human language into code is inherently hard. One reason is that human language is ambiguous by nature. As the famous Winograd test [31] puts it, the sentence "The trophy would not fit in the brown suitcase because `it` was too big", `it` can either mean trophy or suitcase. While a human may be able to decide which one is correct, computers have a harder

time since understanding this sentence requires "the use of knowledge and commonsense reasoning" [32].

There are two general types of ambiguities [33]:

- **Genuine ambiguities**, where a sentence really can have two different meanings to an intelligent listener. An example of genuine ambiguity in the context of Bash Commands is "merge file A with B in folder C". This sentence has at least 2 interpretations: "merge file A with B if B is in folder C" or "merge file A with B and put the result in folder C".
- **Computer ambiguities**, where the meaning is entirely clear to a listener, but a computer detects more than one meaning. A compute ambiguity can occur when multiple parse trees exist for a natural language sentence, such that when the tree is flattened the order of words for input can be undefined.

Both types of ambiguities can affect the performance of translation from natural language to Bash Commands.

4.2 Challenge 2: The Natural Language to Bash Translation Task is Usually a Many-to-Many Mapping

Translation tasks are usually many-to-many mappings, which means there can be multiple correct translations for the same sentence. Moreover, even the sentence itself can have multiple methods of expression. As the size of the dictionary grows, there will be more possible translations for the same input. The process of creating the target sentences requires significant human effort.

Natural language is inherently flexible and Bash Commands can have functional overlap between different utilities. For example, when translating natural language to Bash the phrases `find the word "foo" in file "bar"` and `search in "bar" for "foo"` have the same meaning. Similarly, both `grep -w foo bar` and `cat bar | grep -w foo` are valid translations.

4.3 Challenge 3: Paired English and Bash Commands Data Are Not Easily Accessible

Machine translation models require many training examples. Collecting such a corpus is hard, however, especially for supervised learning that requires paired data (i.e., data with labels) an understanding of both the source and

target languages is needed. Without a large number of training examples, therefore, it may be hard for the model to generalize beyond the small samples in the training set.

Translating natural language to Bash Commands provides a unique challenge in which there are both a large number of English sentences and Bash Commands. However, paired data (i.e., English sentences with the corresponding Bash Commands) are not easily accessible. For paired sources, such as coding help forums like Stack Overflow, the question is usually a detailed description of the command that is summarized succinctly by humans. Writing Bash Commands requires considerable coding skills and is thus hard to crowd-source.

4.4 Challenge 4: Bash Commands Change Environments and Are Generally Computer Specific

Bash Commands are often executed on the command line and are used for file manipulation, search, and application-specific scripting. When these commands are generated in large quantities they often result in deleted files, undefined behavior, and incredibly large searches. This output not only taxes the environment they run on if executed, but can damage the system itself by deleting critical files and directories.

In addition to potentially dangerous behavior, different file systems vary dramatically and humans use a variety of methodologies when organizing their file systems. This results in infinite unique file system configurations, each with different directory structures, permissions, and file names. With both generated Bash Commands and commands scraped from the Internet, a valid execution on one machine does not ensure a valid execution on another. Moreover, what may achieve the desired result on one machine may crash another.

5 Research Questions

Which Deep Learning Architectures Perform Best When Translating Between Natural Language and Bash Commands?

Since there is relatively little literature published on translating natural language to Bash Commands, an important concern is identifying which architectures published in other domains perform best. In particular, Sequence-to-Sequence [34] models have been studied extensively in the context of translations, so we explored their performance on this particular task. These

Table 1 Model performance comparison

Encoder	Decoder	Accuracy	Train	Inference
Transformer	Transformer	**0.522***	1625	0.126
Transformer	RNN	–	–	–
RNN	Transformer	0.486	1490	0.116
RNN	RNN	0.336	**1151***	0.069
BRNN	Transformer	0.495	1411	0.120
BRNN	RNN	0.476	1218	**0.065***

Table 2 The NLC2CMD leaderboard

Team	Model	Data Augment	Accuracy	Power	Latency
Magnum	Transformer	No	**0.532***	682.3	0.709
Hubris	GPT-2	No	0.513	809.6	14.87
Jb	Classifier+Transformer	Yes	0.499	828.9	3.142
AICore	Two-stage Transformer	No	0.489	**596.9***	0.423
Tellina [12]	BRNN (GRU)	No	0.138	916.1	3.242

models consist of two main components: an *encoder* and a *decoder*. The encoder turns the inputs into vectors and the decoder reverses the process. We compared different combinations of encoder-decoder layers, including RNN, BRNN, and Transformer, to translate the natural language to Bash Commands.

Chen et al. [35] discovered that Transformer quality gains stemmed mostly from the Transformer encoder and that RNN decoders often have faster inference times. We therefore mixed and tested different combinations of encoder and decoder types. Table 1 summarizes the performance comparison (measured in seconds) between different model structures.

The results shown in Table 1 indicate that in this particular case, using the Transformer as both an encoder and decoder has the best accuracy.[1] Likewise, the model with an RNN as the decoder can reduce inference time by 50%.

To provide a high-level perspective on how model architecture impacts performance, we analyzed the architectures of the top-performing teams in the NLC2CMD competition. Table 5 shows the Top 4 teams and the baseline model on the NLC2CMD Challenge leaderboard [22]. The Transformer architecture discussed in Section 5.1 was produced from an analysis of our team Magnum's architecture, which won the accuracy competition.

[1]The OpenNMT [36] framework currently does not support a Transformer encoder + RNN decoder.

AICore [22] won the energy track by having the least energy consumption with a two-stage prediction design consisting of two 2-layer Transformers. The first model predicted the template and the second model filled in the arguments. We suspect their small energy consumption is due to smaller model sizes (in contrast, the Magnum teams' model consisted of six layers). However, the gain in less energy consumption also came with a cost of lower accuracy (4.3% decrease).

Team Hubris [22] adopted a fine-tuned ensemble GPT-2 as the language model and achieved second place in accuracy. GPT-2 models are large (usually more than 5 GB) and power-hungry. It is therefore challenging to apply them as a background program running continuously in a terminal to suggest translations of Bash Commands. Another problem with GPT-2 ensembles is that their inference time was prohibitive for real-world deployment, which requires fast response time and low energy consumption to run continuously in the background. Considerable effort is needed to compress and deploy GPT-2 ensembles to compete with other solutions.

Team Jb [22] augmented the training data using back-translation [37] and created 78,000 augmented training samples. They also used the manual pages of Linux Bash Commands [38] to concatenate utilities with corresponding flags and generate an additional 200,000 new samples. Similar to Team AICore, they also used a two-stage model consisting of a classifier for utility prediction and a transformer for command generation. Interestingly, a large number of additional training samples was insufficient to overcome the architectural improvements of other teams.

The results shown in Table 5 provide several key insights:

- Transformer models were the most popular choice. In this task, two-stage models performed worse than a single-stage-and-larger model.
- GPT-2 approaches achieved near state-of-the-art accuracy, but produced much larger models compared to Transformers and had much longer inference times.
- Data augmentation improved accuracy (Team Jb is 1% more accurate than Team AICore) but had less impact than the model structure in this task (with the caveat that the two teams had similar – but not identical – models).

The experiments in the remainder of this paper use Transformer models since they were the best-performing architecture in the NLC2CMD competition.

How do Bash Command Parameters Affect the Performance of Natural Language to Bash Translation?

As discussed in Section 4.4, obtaining training data of paired English and Bash Commands is hard. Without sufficient training data, the model may not be able to learn the entire vocabulary that it must translate to or from. Finding ways of reducing vocabulary size is thus essential to developing more accurate models.

Bash Commands typically consist of three terms: (1) utilities that specify the main goals of the command (e.g., `ls`), (2) flags that provide metadata regarding command execution (e.g., `-verbose`), and (3) parameters that specify directories, strings, or other values that the command should operate on (e.g., /usr/bin). Each utility has a bounded number of flags that can be passed to it. In contrast, parameters have a much larger range of values. Training examples for translating natural language to Bash Commands provide values for the parameters, which can vary significantly between translated examples of the same command.

We hypothesized that including the actual parameter values (such as `ls /usr/bin` and `ls /etc`) from the training examples would vastly increase the overall vocabulary size and decrease model accuracy. Our rationale for this hypothesis is that there are few paired examples of natural language and Bash Commands. Translation models therefore typically perform worse with large vocabulary sizes and limited training data.

To test this hypothesis, we used the English and Bash tokenizers from the Tellina model [12] with our modification. As shown in Figure 1, Bash tokens can be categorized as utilities, flags, and parameters (i.e., arguments, such as a specific path). The English tokenizer decapitalized all the letters and replaced parameters with generic forms. The Bash tokenizer parsed Commands into syntax trees with each element labeled as utility, flag, or parameter.

Our accuracy metric focused mainly on the structure and syntactic correctness of the Bash Command. We therefore replaced all the parameters in Bash with their corresponding generic representations. For example, a folder

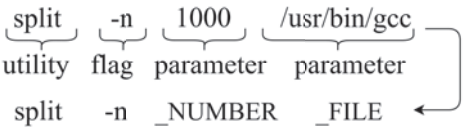


Figure 1 Example of a Bash Command.

Table 3 Parameter replacement

Encoder	Decoder	Accuracy	Accuracy (NP)
Transformer	Transformer	0.509	**0.522***
Transformer	RNN	–	–
RNN	Transformer	0.448	**0.486***
RNN	RNN	0.151	**0.336***
BRNN	Transformer	0.483	**0.495***
BRNN	RNN	0.301	**0.476***

path like `/usr/bin` is replaced with `PATH`. By applying this transformation, the Bash vocabulary size was reduced from 8,184 to 776 tokens and the accuracy of the Transformer models we tested increased by 1.3%. As shown in Table 3, we achieved accuracy and performance increases across all architectures, especially for the ones with less accuracy.

How to Expand the Amount of Available Bash Command English Language Pairs Without the Hiring of External Freelancers?

As discussed earlier, further innovation and developments when attempting to solve the natural language to Bash Command translation problem are severely restricted by the limited number of command-natural language pairs provided in the original dataset. This question not only deals with the most effective way of adding new and valid Bash Commands, but also deals with creating corresponding natural language pairs.

Scraping from online forums, such as StackOverflow, effectively gathers commands that could be added to the dataset, but yield significant problems when dealing with invalid commands, duplicate commands, and commands of different programming languages. Another potential approach could be to augment the training data available, making minor changes to commands like the removal of a flag. However, this approach also poses challenges in differentiating between similar commands and adds minimal diversity of function to the dataset.

We decided to create a Bash Command generator and use manual page data to synthesize entirely new Bash Commands. This approach allowed us to curate the synthesized dataset, maintaining similarities to the existing dataset, while still introducing informative data points.

To create corresponding natural-language components, we used a back-translation model with a transformer architecture. Transformer models have proven extremely effective with summarization tasks, which our back-translation was beginning to closely resemble.

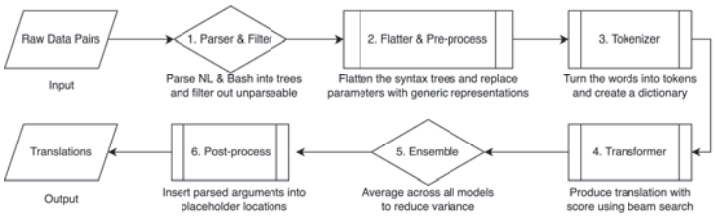


Figure 2 Pipeline of the NLC2CMD Workflow.

5.1 Summary of the Highest Performing Architecture

We tested several different data processing, architectural, and post-processing strategies, as discussed above. We now describe the best-performing model that we tested on the NLC2CMD competition data. Although this model will be improved by subsequent work, it provides a starting point for researchers focusing on natural language to Bash Command translation. In particular, our results show that the Transformer model is a robust foundation for future research in this area.

Our Transformer model pipeline was built from the following six steps shown in Figure 5 and described below:

1. **Parsers and filters** – The paired raw data first go through different parsers that convert English sentences and Bash Commands into syntax trees (data that cannot be parsed are removed).
2. **Flatten and pre-process** – The syntax trees are flattened and the parameters are replaced with their generic representations.
3. **Tokenizer** – The flattened sentence pairs are tokenized and dictionaries are created for English sentences and Bash Commands.
4. **Transformer models** – Tokenized sentences are fed into Transformer models and Beam Searches are enabled to produce multiple translations.
5. **Ensemble** – The best-performing models on the validation dataset are chosen to create an ensemble.
6. **Post-process** – The translations produced by the ensemble model are post-processed by removing the placeholder arguments and inserting the values originally removed by the parser.

5.2 Parsing and Tokenization

For our investigation, we used both the NLC2CMD dataset (which contains 10,347 pairs of English sentences and their corresponding Bash Commands)

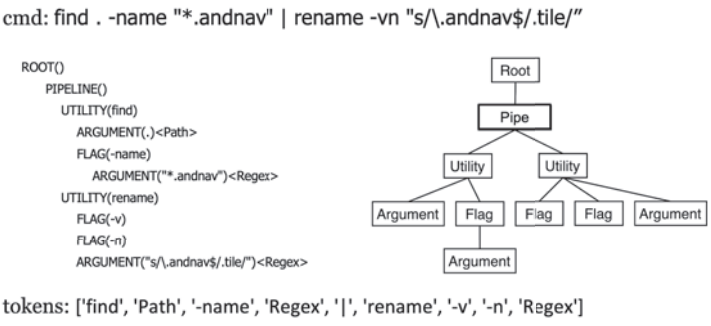


Figure 3 Visualization of the Tokenization process.

and our generated dataset (which consisted of 71,705 English sentence-Bash Command pairs). Of the 10,347 pairs of data in the original dataset, 29 had grammar issues and were therefore excluded. The size of this public dataset was relatively small in the natural language processing research field[2] and the goal for data processing was to create a small word vocabulary and utilize as much data as possible. Our generated dataset was significantly larger, so we shifted our focus to achieving higher quality commands, as opposed to using as many of the generated commands as possible.

Bash Commands can be complex and nested, as shown in Figure 3. This structure helps explain why programmers may find it hard to create – or even comprehend – Bash Commands, thereby motivating the need for a customizable parser. Bash Commands can also be piped, which means the Commands may consists of multiple parts, with the output of the former part been the input for the latter one.

We built our parser atop the Tellina [12] parser developed based on Bashlex [39] in prior work. This parser can parse a Bash Command into an abstract syntax tree (AST) composed of utility nodes, each of which may contain multiple corresponding flags and parameters. During the tokenization stage, utilities and flags are kept "as is" and parameters are categorized and replaced with `_NUMBER`, `_PATH`, `_FILE`, `_DIRECTORY`, `_DATETIME`, `_PERMISSION`, `_TIMESPAN`, `_SIZE`, with the default option of `_REGEX`.

Natural language sentences are pre-processed by filtering out the stop words (e.g, "a", "is", "the", which carries little meaning). The remaining words are then decapitalized and lemmatized (preserving the common base form) to create a relatively smaller dictionary mapping.

[2] In comparison, WMT-14 en-de (a popular dataset for machine translation benchmark) has 4.5 million sentence pairs.

Our generator used a different parameter categorization strategy than the parser. The goal was to better align with the interpretation of the manual pages and the generation of high-quality commands. Parameter categorizations were similar, however, and the generator categorization was easily converted to the representation of the parser for training and inference with the transformer-based model.

5.3 Model Details

The model with the highest accuracy used a Transformer as both the encoder and the decoder, as shown in Figure 4. The encoder and decoder each consisted of six layers. The model was trained for 2,500 steps and used an ensemble of the four top-performing single models.

The first positional weight was set to 1.0 and the rest of the weights were set to the exponential of beam scores capped by 0.5. We focused on training an efficient and robust model that can be deployed easily. The need to modify the network structure was therefore relatively low. Instead of FairSeq [40] (which allows users to modify the low-level network structure), we chose OpenNMT [36], which is an open-source neural sequence learning framework to implement our Transformer model.

We found that the Transformer model is sensitive to learning rate and larger batch sizes will produce better results. The detailed training hyperparameters are available on our GitHub repository [41]. Likewise, the guiding principle behind our tuning strategy is derived from Popel et al. [42].

We trained our model on 2 Nvidia 2080 Ti Graphic cards with 64GB memory. Our model achieved 53.2% accuracy on the hidden test dataset for

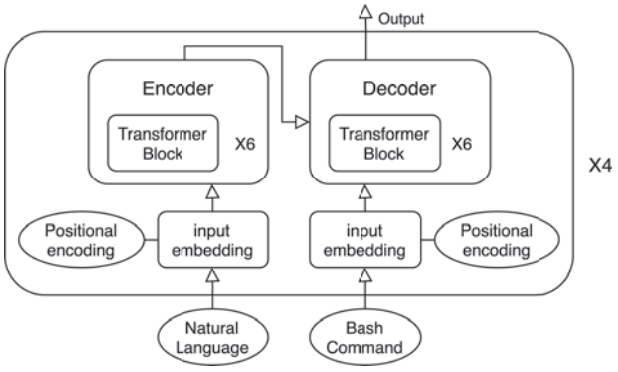


Figure 4 Model structure.

the NLC2CMD competition and had top performance in both inference time and energy consumption. We addressed challenge 4.1 by masking out specific parameters. To limit ambiguity, the dataset itself also restricted the natural language description to a single sentence and the Bash Command to a single line [9]. When the dataset was collected, the same Bash Command was paired with many English descriptions to increase language diversity [9], thereby addressing Challenge 4.4.

5.4 Post-processing

The initial results of our model translation contained the placeholders inserted by the parser described above. This insertion was done to create a smaller dictionary mapping and improve the accuracy of the model. This result, although accurate, lacks practicality and many of the commands are confusing to humans and are far from executable. For example, the inclusion of _REGEX in a command translated from the English language is vague and hard for anyone looking to use the workflow as a tool. To address this issue, we added post-processing of the translated commands into our workflow. Both inexperienced users and machines themselves can make use of executable commands, easily running them in a terminal, yet commands nested with vague placeholders are less useful.

Using the parser from the pre-processing of the natural language, we extracted the parameters provided in the original corpus and used them to fill out placeholders in the created translation. Many translated commands had a perfect one-to-one replacement with the extracted parameters replacing all of the placeholders to create executable Bash Commands. Some other commands, however, contained more placeholders than parameters extracted from the parser. which resulted in partially replaced commands. Although these commands were not executable, they were also not implicitly incorrect translations.

For example, the description "remove all files in the current directory with a specific inode number" may have translated to `find . -inum Quantity exec rm .` In this translation, the specific inode number represented by the `Quantity` placeholder was not replaced as it never appeared in the original description. This demonstrates how the creation of partially replaced, non-executable commands can still be accurate translations, such that efforts towards partial replacement are worthwhile. This last step in the workflow increases usability and practicality surrounding the translation pipeline by converting Bash Command templates to executable or nearly executable commands that better resemble the purpose described in the natural language.

6 Corpus Construction

We were able to construct a corpus of "Bash Command and natural language" pairs six times larger than that of the original dataset. We created this large corpus by developing a generator to synthesize millions of Bash Commands, which were later validated and scaled. We then fed these commands through our back-translation model to create the corresponding natural language pairs.

6.1 An Updated Pipeline

In our updated pipeline we include the most successful aspects of our prior pipeline to generate an entirely new dataset and then train and test our top-performing model on our new dataset. Our state-of-the-art transformer-based model has proven effective in machine translation, so we decided to incorporate it both in our dataset generation and our training. Our updated pipeline, as demonstrated in Figure 6, consisted of the steps described below:

1. **Manual Page Scraping** – We scraped data from the manual pages to determine the syntax usage of 38 utilities. We also determined the flags associated with each utility and the categorization of the parameter, if any, associated with each of those flags.
2. **Generation** – With the syntactical structures, flags, and arguments for each utility, we generated over 1 million Bash Commands from different combinations of flags and piped commands.
3. **Validation** – The generated commands were then replaced with actual arguments. For example, `[File]` was replaced with `temp.txt`. These commands were executed on a virtual machine and we discarded all commands that did not execute successfully with exit statuses of zero within a given time frame.
4. **Scaling** – The validated commands were then converted into a form understandable by the parser, parsed, and scaled. This step involved preserving a similar proportion of commands with the `find` utility to the original dataset and ensuring there was a diversity of other utilities in the new dataset. Likewise, we discarded commands of over-represented utilities and commands that were parsed incorrectly by the parser.
5. **Back-translation** – The validated commands were then converted into a form understandable by the parser and fed to the back-translation model. This model was the same transformer-based model used on the original dataset, except trained in the reverse direction, using Bash Commands to

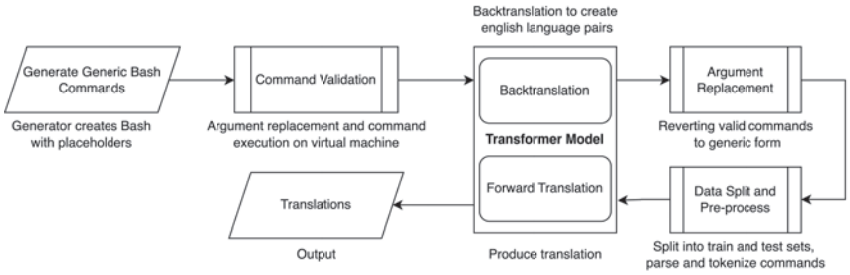


Figure 5 Updated Pipeline of the Dataset generation and translation.

predict natural-language sentences. This step created the corresponding natural language pairs for the generated dataset.

6. **Forward translation** – The new dataset was then split into training and testing and used to train and evaluate the model. For the validation, the best-performing models on the validation dataset was chosen to create an ensemble.

6.2 Bash Command Synthesis

Our generation stage involved scraping manual page information and assembling together commands from individual components. We used data gathered from the Linux manual pages to form syntax structures to help our generator understand the relationship between the different components from which commands are formed. Bash Commands generally consist of utilities followed by flags and arguments, although complexity can increase dramatically with the introduction of piped and nested commands. With sophisticated web-scraping techniques and some manual oversight, generated a mapping of utilities to their corresponding arguments and flags. Likewise, each flag had corresponding arguments, as the introduction of flags often increased the number of arguments in a command.

We also scraped the syntax for each utility, creating templates to outline the context in which each utility is used and the order in which the flags and arguments appeared. In total, we created mappings and templates for the 38 of the most common utilities appearing in the original NL2Bash dataset.

With the collection of this data, we generated thousands of commands with combinations of zero to three flags present in each command. The number of potential commands our generator is capable of producing is in

the billions. However, we limited the number of commands produced for the following reasons:

- **Quality preservation**. While generating commands that include large numbers of pipes and flags may yield many valid and executable commands, these commands are rare and thus do not commonly appear in the original training dataset. Other characteristics we tried to preserve from the original training dataset when synthesizing our own commands were general similarities in utility distributions and the ratio of commands with a pipe to those without.
- **Practicality**. In both of our validation and back-translation processes described below, the amount of time required to process the dataset scales with the number of commands in the dataset. After several hundred thousand commands, it becomes less practical to devote further resources to additional command generation.

In the scaling stage, we scaled the command generation to generally resemble the utility distribution of the original NL2Bash dataset. For example, the original dataset consisted of 63.44% commands that began with the utility `find`, so we scaled the number of the `find` commands generated to represent a similar percentage of our generated commands.

We attempted to do this scaling for all generated utilities, although we achieved varying results. The original dataset consisted of 117 different utilities, while our generator only supported 38, as shown in Table 4. Moreover, many of these 38 utilities had poor or inconsistent documentation in the manual pages, making it hard to accurately collect all available flags and arguments and generate enough commands to match the distributions desired.

Another limiting factor in the distribution matching was the difficulty of generating valid commands for certain utilities. As described in the next section, every generated command was later validated to determine whether or not to include it in the dataset. The likelihood of commands of certain utilities being deemed invalid was significant. Generating large quantities of

Table 4 Command generation process

	Generation	Validation	Scaling
Utility Count	38	35	35
Non-piped Commands	570,436	60,926	38,557
Single-piped Commands	500,000	81,787	33,148

commands for those utilities therefore hindered their representation in the dataset.

Some utilities had few available flags and a large number of duplicate appearances in the original dataset. For example, the command `cd [Directory]` appeared 13 times in the training data as it originally appeared without placeholders. In the generated dataset, however, there were no duplicate commands, so the command only appeared once, thereby limiting the representation of that utility in the generated data.

The generated dataset not only included commands with a single utility but piped commands, as well. These commands consist of multiple utilities or commands that were concatenated, allowing the sharing of information during execution. Implementing support for piped commands involved analyzing the training data for common utility pairs piped together, generating commands independently for each utility, and joining them together with a pipe symbol.

For example, the most popular utility pair in the training data was shown to be (`find`, `xargs`). In particular, of the piped commands, a command with the `find` utility was often followed by an `xargs` instruction piped afterward. Of the commands in the original training dataset, 31.36% of them contained one or more pipes, with the mean number of pipes in each piped command being 1.45.

Due to the increased complexity in piped command support, we included some simplifications in the generation of piped commands. In particular, 70.33% of the piped commands in the training data only contained a single pipe, so we only supported commands with one pipe. Moreover, in contrast to commands without pipes, the utility distribution matching for piped commands encompassed fewer utilities. 43.71% of the single pipe commands in the training data consisted of `find` commands that were followed by `xargs`, `grep`, and `sort` commands, so we made the decision to support these combinations with piped command generation.

The inclusion of piped commands in the generated dataset dramatically increased the number of available commands to generate. The number of potential commands to generate scaled exponentially when pipes were introduced, yet this further pressured the validation process that executed all the generated commands sequentially. As a result, we limited the generation of piped commands to combinations from 500 different `find` commands concatenated to 1,000 different `xargs`, `grep`, and `sort` commands, totaling at 500,000 commands synthesized in the first stage.

6.3 Bash Command Validation

To ensure the validity of the generated commands, each command was replaced with valid placeholder arguments and executed in an isolated environment. An example is `cd [Directory]` being converted to `cd abc` with the directory `abc` being available on the machine. To protect against undefined behavior, commands were executed in a virtual machine environment.

The commands were executed in series and each exit status was measured by the program. The commands that executed to completion and returned with exit statuses of zero were set aside, whereas those with non-zero exit statuses were discarded. The valid commands set aside were then converted back to generic commands, this time using the placeholders available in the original NL2Bash dataset. To prevent hanging, the commands validation script had a timeout of 0.5 seconds for each command before deeming it invalid. However, the percentage of commands deemed invalid due to timeouts was insignificantly small.

The percentage of commands that executed to completion with return statuses of zero varied dramatically for different utilities. Notably, commands like `grep` and `ls` were less prone to errors and approximately 50% of them were able to complete execution with exit statuses of zero. `find`, the most common utility in the training data, had a validity rate of 30.4%. Two utilities, `rev` and `rename`, had no generated commands that ran with exit codes of zero and were therefore removed from the generated dataset entirely. Overall, 13.3% of the generated commands were deemed valid with validity rates of 10.7% and 16.4% for the non-piped and piped commands, respectively.

An exit status of zero does not guarantee a high-quality command. Commands can complete execution while still not achieving any change in the environment. For example, a directory change command like `cd .` might

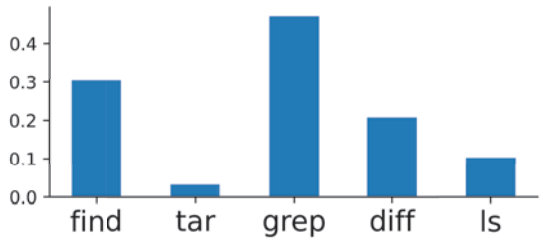


Figure 6 Valid command rates for generated commands.

execute correctly, yet is a highly impractical command as it keeps the user in the same directory they started in.

Moreover, commands that fail the validation stage are not inherently incorrect. Generated commands are generic and placeholders are replaced for validation, so command failure can be a result of argument replacement. The virtual machine may not have the same folders, libraries, and files that are manipulated in a given command, which in some cases resulted in failure. While execution is an efficient way to discard many incorrectly generated commands, it does not correctly classify command validity in all cases.

6.4 English Text Synthesis

To generate the corresponding natural language for our generated Bash Commands, we used the same transformer-based model used in our original translation task. This model architecture proved extremely effective in machine translation tasks, so we reused this architecture for back-translation as opposed to another lower-performing model. In this case, we trained the model using data from the original NL2Bash dataset, but in the opposite direction, *i.e.*, attempting to predict natural language from Bash Commands.

Once the model was trained, we used inference to predict the corresponding natural language for each command in the generated dataset. This completed the natural language component for every validated Bash Command and concluded the dataset generation process.

6.5 Corpus Description and Statistics

In total, our corpus contained 71,705 valid Bash Commands with corresponding English text. 69.9% of these commands began with the `find` utility and the rest were distributed across 34 other utilities, which totals 35 utilities represented. The eight most popular utilities were `find`, `tar`, `grep`, `diff`, `ls`, `file`, `du`, and `cp`.

All our generated commands contained between zero and three flags for each utility within the command. Generally, commands without a pipe contained one utility and commands with a pipe contained two, but in rare instances for specific utilities, like `xargs`, nested commands supported the inclusion of more utilities. As a result, the generated commands contained between one and four utilities, with the vast majority containing just one or two. Of the generated commands, 33,148 (46.22%) of them contained a single pipe, while the rest contained zero pipes. Every command generated

was executed in a virtual machine command-line environment and was able to complete execution with an exit status of zero within 0.5 seconds.

6.6 Data Quality

To maximize the quality of the generated commands, we intentionally chose not to use the same argument-type placeholders as the parser when generating commands. As described above, the parser defaulted to classifying parameters as `_REGEX`, which is a difficult placeholder for executable command generation since it can take many forms.

Instead, we used 15 argument-type placeholders, many similar to those from the parser, but generally centered around our goal of creating valid and executable commands. We wanted to choose argument-type placeholders that were as specific as possible, while still easily scraped and classified from manual pages in an automated fashion. As opposed to one argument type `_REGEX`, we included `Pattern`, `FormattedString`, and `Separator` to differentiate between the different types of regular expressions and improve the particularity of the generated commands. This decision was possible because the manual pages also differentiated between these argument types, so classifying these flags with these corresponding argument types based on keywords in the manual pages required no significant extra work.

Although our strategy for dealing with parameters and placeholders differed at the command generation stage, we did not want this inconsistency to propagate to our translation. The parser and its corresponding placeholders were shown effective in preprocessing, so after command generation we converted the placeholders to match those of the parser to include in our synthesized dataset. This conversion was generally straightforward, with each of our chosen placeholders mapped to only one parser placeholder.

Moreover, our validation of all the generated commands through injection of actual arguments into the placeholders and execution in a virtual machine environment ensured a high level of quality for our generated commands. Of our 570,476 no-pipe commands generated, 60,926 commands (10.7%) of our generated commands executed to completion with exit statuses of zero.

This level of validation was not performed on the initial dataset. We found that many of the commands provided in the original NL2Bash dataset were unable to execute to completion with exit statuses of zero in our environment. This result suggests the possibility of Bash Command generation eventually yielding a higher percentage of valid commands than those manually verified by expert freelancers.

6.7 Comparison to Existing Dataset

As mentioned in Section 6.2, we attempted to match the distribution of the training data when determining the ratio in which we generated commands for different utilities. Commands with the `find` utility amounted to the majority of the training commands and were the limiting factor when determining the size of the generated dataset. Despite these efforts, there were key differences between the existing and generated dataset, as shown in Table 5.

The original dataset contained a large diversity of commands, with 117 different utilities present. Since our generator only supported 38 utilities, many utilities represented a larger proportion of the dataset to account for those that were missing. For most of the included utilities, the quantity of generated commands for a given utility drastically outnumbered the number of commands for the utility in the original dataset. These utility distributions of the two datasets are shown in the Figures 7 and 8, where the most common utilities in the generated dataset are plotted alongside those of the original dataset.

There are several reasons for the differences in the composition of the two datasets. For example, the existing dataset contained many duplicate commands or commands of the same structure applied to different files, directories, or other arguments. In contrast, this level of duplication did not occur for synthesized dataset since every Bash Command generated is unique. This duplication also resulted in popular commands with limited use cases, such as `cd [Directory]` being underrepresented in our generated commands with respect to the existing dataset.

Table 5 New vs. existing dataset

Category	Original	Generated (Raw)	Generated (Valid)
Total Commands	10,348	1,070,436	71,705
Piped Commands	3246	500,000	55,931
Distinct Utilities	117	36	36

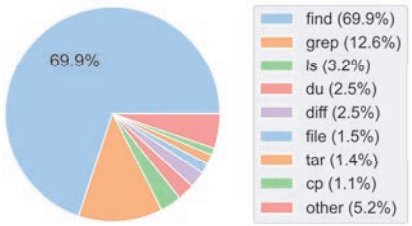


Figure 7 Utility distribution in the generated dataset.

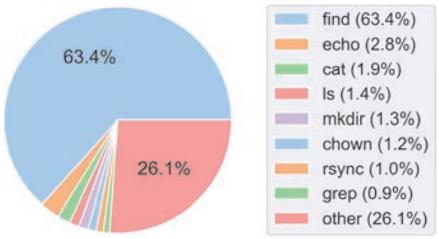


Figure 8 Utility distribution in the original NL2Bash dataset.

More generally, the original dataset contained large quantities of popular commands and underrepresented more obscure commands or those with unpopular flags due to the data gathering strategy of searching online forums, as shown in Figure 8. This strategy resulted in a heavily biased dataset and little-to-no coverage of some powerful – yet uncommon – flags. Our generated dataset gave no preference to popularity and treated all flags equally, although those less likely to result in errors or time-consuming execution times were often filtered in validation stages. As a result, our dataset was significantly more diverse, with many unusual flag combinations represented that were not in the original dataset.

Moreover, our synthesized dataset included up to three flags for each utility in the command and at most a single pipe within the command. This configuration meant that every generated command had a maximum of two utilities and six flags, excluding a small minority of nested commands. Although the majority of the commands in the existing dataset matched this demographic, many commands in the training data had several pipes or contained more than three flags proceeding a utility. This configuration meant that although the generated dataset included a large number of diverse commands, the commands were generally shorter in length and minimally complex.

The last key difference involved the validity of the commands. Since the entirety of the generated dataset went through the validation process and only the commands deemed valid were kept, 100% of the resulting dataset was valid. When the original dataset was run under the same conditions, only 2,360 commands were able to execute with exit statuses of zero within the 0.5-second time frame, for a validity rate of 22.8%.

In summary, the analysis above shows that although the two datasets were similar in many ways including the utility composition, there were key differences in utility and flag diversity, as well as validity rates.

7 Metrics and Error Analysis

This section discusses different metrics and performances of the transformer-based Magnum model on both datasets. Section 7.1 describes the accuracy metric and proposes an improved energy metric, Section 7.2 summarizes the accuracy performance of the Magnum model on the new NL2CMD dataset, and Section 7.3 analyses the distribution of different error types on the original dataset.

7.1 Metrics

Below we describe the accuracy metric and proposes an improved energy metric.

Accuracy: The ideal metric for an evaluation would check if the predicted Bash Command produces the same result as the reference answer. That metric is not practical, however, since establishing a simulated environment for 10K variant situations is beyond the scope of this paper. Instead, our scoring mechanism specifically checks for structural and syntactic correctness that "incentivizes precision and recall of the correct utility and its flags, weighted by the reported confidence" [22]. The metric first defines two terms: Flag score S_F^i and Utility score S_U^i.

As shown in Equation (1) [22], the flag score is defined as twice the union of reference flags and predicted flags number minus the intersection, scaled by the max number of either reference flags or predicted flags.

$$S_F^i(F_{\text{pred}}, F_{\text{ref}}) = \frac{1}{N}\Big(2 \times |F_{\text{pred}} \cap F_{\text{ref}}| - |F_{\text{pred}} \cup F_{\text{ref}}|\Big) \tag{1}$$

The range of flag scores is between -1 and 1.

As shown in Equation (2) [22], the utility score is defined as the number of correct reference utilities scaled by capping flag score between 0 and 1, minus the number of wrong utilities, scaled by the max number of either reference utilities or predicted utilities.

$$S_U = \sum_{i \in [1,T]} \frac{1}{T} \times \left(|U_{\text{pred}} = U_{\text{ref}}| \times \frac{1}{2}\Big(1 + S_F^i\Big) - |U_{\text{pred}} \neq U_{\text{ref}}|\right) \tag{2}$$

By summing all the utility scores within a predicted command, the range of normalized utility scores is between -1 and 1.

Energy: The measurement and reporting of energy consumption of natural language programming (NPL) models is a relatively new phenomenon [43] [44]. As Henderson et al. [45] pointed out, part of the reason stems from the complexities of collecting the result. In particular, according to Appendix B of Henderson et al. [45], out of 100 NeurIPS papers from the 2019 proceedings, only 1 measured energy consumption in some way, whereas 45 measured runtime performance.

To address this gap, the NeurIPS 2020 conference recommended "energy" as a more direct way of measuring environmental impact. We found the current energy metric used by the NL2CMD competition was not ideal, however, since it used estimated attributable power draw (mWatts) to compute scores. This metric disproportionately punished models with less inference time.

For example, the GPT-2 model with an inference time of 14.87 seconds should have consumed a huge amount of energy (considering the model size). On the leaderboard shown in Table 5 the power metric is even less than the baseline, which is a much smaller model (GRU) and the inference is 3.24 seconds. Moreover, energy$_{mWh}$ can be easily affected by trivially extending inference time. For example, by simply sleeping 3 seconds after each batch, the performance of a test submission can be improved from 682 to 88 on the leaderboard. A potential fix would be to measure the total energy consumed instead of the power since it punishes both bigger model size and longer inference.

Validity: Another metric measures the quality of Commands that datasets used. Our validity rate metric measured the percentage of commands that were able to execute to completion with exit statuses of zero within a 0.5-second time frame when replaced with standard replacement values. Although this metric is not entirely reliable due to the complex nature of computers behaving differently and containing different file systems, it provided valuable insight into the general correctness of the commands in the dataset.

Commands that *passed* the validity test demonstrated a baseline level of error aversion and proved they did not result in undefined behavior under the given conditions. Commands that did *not pass* the validity test were not implicitly incorrect, however, as they could have executed correctly on a different machine or simply required longer than the allotted 0.5 seconds to execute.

Table 6 Comparison of model performance across datasets

Training Dataset	Test Dataset	Accuracy Score
NL2Bash	NL2Bash	52.3%
NLC2CMD	NLC2CMD	31.6%
NLC2CMD	NL2Bash	−13%
NL2Bash	NLC2CMD	−6.67%
NLC2CMD+NL2Bash	NL2Bash	48%
NLC2CMD+NL2Bash	NLC2CMD	31.7%

7.2 Synthesized Dataset Results

We split our generated dataset 80/20 into training and testing sets, respectively, before training our transformer-based model on the training dataset. We then evaluated our results and achieved an accuracy score of 31.63%, as shown in Table 6. This relatively low score demonstrates the difficulty of the dataset. We conjecture this result occurred because the dataset contained utility-flag combinations that were not present in the original dataset. These combinations resulted in inaccurate back-translations for the natural language components, which made predictions for the model extremely hard.

We also suspect over-fitting for the model trained on the NL2Bash dataset due to its relatively small size. We found that the model cross-trained on both datasets demonstrated good performance on both test sets, while the model trained on a single dataset performed poorly on the test set of another dataset. This result again highlights the importance of having large and diverse dataset for increasing model robustness.

7.3 Error Analysis

Previous research [9] listed the top three causes of incorrect predictions as sparse training data, utility errors, and flag errors. Since sparse training data is a subjective metric, we only analyzed the incorrect utility and flag predictions. We used a separate, independently-created testing dataset of 1,867 samples (previous work manually analyzed 100 samples from the dev dataset collected the same way as the original training dataset) from the original training dataset and evaluated the accuracy results in more detail.

Figure 9 shows that over two-thirds of all errors are utility errors, so the variety of flags is less significant than having enough data for each utility. This result shows why our new dataset resulted in a significantly lower accuracy score. In particular, utilities that are less common in the original dataset have larger quantities of commands.

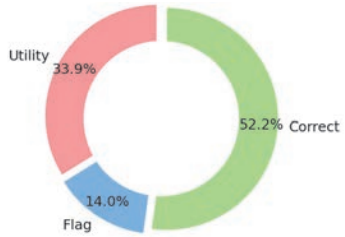


Figure 9 Percentage of utility and flag errors on original dataset.

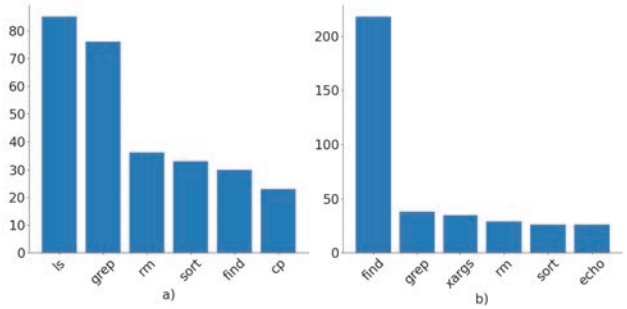


Figure 10 (a) Distribution of reference utilities that are wrongly predicted. (b) Distribution of wrongly predicted utilities.

Figure 10 shows that among the top six incorrectly predicted utilities, `ls` and `grep` are the most frequently confused with `find`. This confusion was expected since the functionality of these three utilities overlaped significantly and were among the most frequently used Bash Commands. By manually examining the incorrect predictions, we also found that these three utilities appear in many piped commands, which helps explain the large proportion they comprised in all the incorrectly predicted utilities. Our synthesized dataset contained both larger absolute and relative quantities of piped commands, so this result further underscored the difficulty of the model for predicting them correctly.

8 Concluding Remarks

This paper presented several key findings for the semantic parsing research community. We first described an updated workflow for a state-of-the-art machine translation model to generate accurate and practical commands. We then introduced post-processing to replace placeholders in translated Bash

Commands with the original parameters provided in the natural language. Our final contribution was a new dataset generated from scratch and an accompanying method for generating additional data.

Our work provides essential foundations for building an automated system that translates natural language to Bash Commands. We were the first to (1) create an entirely valid Bash Command dataset from scratch and (2) provide a baseline accuracy of 31.6% for translating natural language to Bash Commands on the new dataset. Our code is available in GitHub repository [46]. The following is a summary of our lessons learned gleaned from the research project:

- **It is feasible to synthesis a large dataset of Bash Commands and corresponding English pair by adopting back-translation.** Generation from scratch is a major milestone and provides significant advantages over prior augmentation strategies. Our approach provided new opportunities for the generation of further natural language to computer code datasets and improving the effectiveness of Bash Command machine translation.
- **To make translated commands practical they must be executable, therefore validity testing is important.** The conversion from a Bash Command template to an executable (or nearly executable) command shows both progress and promise of the usability and practicality of our translation pipeline. A more complete and streamlined process of converting natural language to valid, executable commands will become a larger focus as model accuracy continues to improve.
- **It is necessary to create a hold-out dataset[3] that is sourced in a different way to test model generality.** Although our dataset was six times larger then the original dataset and more diverse, the model exhibiting good performance on the test set failed to generalize to the original dataset and vice versa. As a result, the current datasets of Natural language to Bash Commands are still relatively small and insufficiently diverse to make robust model that generalize well to a hold-out set. This result could yield further developments and testing of different models to maximize performance on both our dataset and the original dataset.

[3] A hold-out dataset is sourced differently from the original data, so it is more challenging compared to test-set, which is sourced the same way as the training data and likely has the same distribution.

References

[1] Raymond J Mooney. Semantic parsing: Past, present, and future. In *Presentation slides from the ACL Workshop on Semantic Parsing*, 2014.

[2] Quchen Fu, Zhongwei Teng, Jules White, and Douglas C Schmidt. A transformer-based approach for translating natural language to bash commands. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1245–1248. IEEE, 2021.

[3] Shikhar Bharadwaj and Shirish Shevade. Explainable natural language to bash translation using abstract syntax tree. In *Proceedings of the 25th Conference on Computational Natural Language Learning*, pages 258–267, 2021.

[4] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. *arXiv preprint arXiv:1802.08979*, 2018.

[5] Quchen Fu, Zhongwei Teng, Jules White, and Douglas C. Schmidt. A transformer-based approach for translating natural language to bash commands. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1245–1248, 2021.

[6] J. Sammet. The use of english as a programming language. *Commun. ACM*, 9:228–230, 1966.

[7] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

[8] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM, 2018.

[9] Xi Victoria Lin, C. Wang, Luke Zettlemoyer, and Michael D. Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. *ArXiv*, abs/1802.08979, 2018.

[10] Jonathan Berant, A. Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *EMNLP*, 2013.

[11] Xuliang Liu and H. Zhong. Mining stackoverflow for program repair. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 118–129, 2018.

[12] Xi Victoria Lin. Program synthesis from natural language using recurrent neural networks. 2017.

[13] Tomas Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, 2010.

[14] Jichuan Zeng, Xi Victoria Lin, S. Hoi, R. Socher, Caiming Xiong, Michael R. Lyu, and Irwin King. Photon: A robust cross-domain text-to-sql system. *ArXiv*, abs/2007.15280, 2020.

[15] J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.

[16] A. See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *ACL*, 2017.

[17] Ursin Brunner and Kurt Stockinger. Valuenet: A neural text-to-sql architecture incorporating values. *ArXiv*, abs/2006.00888, 2020.

[18] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *NIPS*, 2015.

[19] Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. Incorporating external knowledge through pre-training for natural language to code generation. *ArXiv*, abs/2004.09015, 2020.

[20] Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *EMNLP*, 2018.

[21] J. Chung, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *ArXiv*, abs/1412.3555, 2014.

[22] Mayank Agarwal, T. Chakraborti, Q. Fu, David Gros, Xi Victoria Lin, Jaron Maene, Kartik Talamadupula, Zhongwei Teng, and J. White. Neurips 2020 nlc2cmd competition: Translating natural language to bash commands. *ArXiv*, abs/2103.02523, 2021.

[23] Isaac Caswell and Bowen Liang. Recent advances in google translate, 2020.

[24] Mike Schuster and K. Paliwal. Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.*, 45:2673–2681, 1997.

[25] Bash reference manual, 2020.

[26] Shikhar Bharadwaj and Shirish Shevade. Explainable natural language to bash translation using abstract syntax tree. In *Proceedings of the 25th Conference on Computational Natural Language Learning*, pages 258–267, Online, November 2021. Association for Computational Linguistics.

[27] Xuan-Phi Nguyen, Shafiq Joty, Kui Wu, and Ai Ti Aw. Data diversification: A simple strategy for neural machine translation. *Advances in Neural Information Processing Systems*, 33:10018–10029, 2020.

[28] Yuekai Zhao, Haoran Zhang, Shuchang Zhou, and Zhihua Zhang. Active learning approaches to enhancing neural machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1796–1806, Online, November 2020. Association for Computational Linguistics.

[29] Mayank Agarwal, Kartik Talamadupula, Fernando Martinez, Stephanie Houde, Michael Muller, John Richards, Steven I Ross, and Justin D Weisz. Using document similarity methods to create parallel datasets for code translation. *arXiv preprint arXiv:2110.05423*, 2021.

[30] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[31] H. Levesque, E. Davis, and L. Morgenstern. The winograd schema challenge. In *KR*, 2011.

[32] H. Levesque. On our best behaviour. *Artif. Intell.*, 212:27–35, 2014.

[33] Ernest Davis. Notes on ambiguity.

[34] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[35] M. Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, G. Foster, Llion Jones, Niki Parmar, M. Schuster, Zhi-Feng Chen, Yonghui Wu, and Macduff Hughes. The best of both worlds: Combining recent advances in neural machine translation. In *ACL*, 2018.

[36] G. Klein, Yoon Kim, Y. Deng, Jean Senellart, and Alexander M. Rush. Opennmt: Open-source toolkit for neural machine translation. *ArXiv*, abs/1701.02810, 2017.

[37] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, M. Srivastava, and Kai-Wei Chang. Generating natural language adversarial examples. In *EMNLP*, 2018.

[38] Free software foundation (2018) linux, 2018.

[39] Idan Kamara. Bashlex, 2014.

[40] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, S. Gross, Nathan Ng, David Grangier, and M. Auli. fairseq: A fast, extensible toolkit for sequence modeling. *ArXiv*, abs/1904.01038, 2019.

[41] Magnum-nlc2cmd, 2020.

[42] M. Popel and Ondrej Bojar. Training tips for the transformer model. *The Prague Bulletin of Mathematical Linguistics*, 110:43–70, 2018.
[43] Emma Strubell, Ananya Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp. *ArXiv*, abs/1906.02243, 2019.
[44] Qingqing Cao, Aruna Balasubramanian, and Niranjan Balasubramanian. Towards accurate and reliable energy measurement of nlp models. *ArXiv*, abs/2010.05248, 2020.
[45] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. Towards the systematic reporting of the energy and carbon footprints of machine learning, 2020.
[46] Bash gen, 2022.
[47] Hugh Chen, Scott Lundberg, and Su-In Lee. Checkpoint ensembles: Ensemble methods from a single training process. *ArXiv*, abs/1710.03282, 2017.
[48] Katherine Lee, Orhan Firat, A. Agarwal, C. Fannjiang, and David Sussillo. Hallucinations in neural machine translation. 2018.
[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, L. Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.
[50] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.
[51] Yonghui et al. Wu. Google's neural machine translation system: Bridging the gap between human and machine translation. *ArXiv*, abs/1609.08144, 2016.
[52] A. Radford, Jeffrey Wu, R. Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.
[53] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks : the official journal of the International Neural Network Society*, 61:85–117, 2015.
[54] James Bennett and Stan Lanning. The netflix prize. 2007.
[55] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation, 2017.
[56] Robert Chatley, Alastair Donaldson, and Alan Mycroft. *The Next 7000 Programming Languages*, pages 250–282. Springer International Publishing, Cham, 2019.
[57] M. Agarwal, Jorge J. Barroso, T. Chakraborti, Eli M. Dow, Kshitij P. Fadnis, Borja Godoy, and Kartik Talamadupula. Clai: A platform for ai skills on the command line. *ArXiv*, abs/2002.00762, 2020.
[58] M. Boot. Redundancy in natural language processing. 1978.

[59] C. Hoare. Hints on programming language design. 1973.
[60] Neurips 2020 competition track, 2020.
[61] J. Vig. A multiscale visualization of attention in the transformer model. *ArXiv*, abs/1906.05714, 2019.
[62] Thomas G. Dietterich. Ensemble methods in machine learning. In *Multiple Classifier Systems*, 2000.
[63] Sam Wiseman and Alexander M. Rush. Sequence-to-sequence learning as beam-search optimization. In *EMNLP*, 2016.
[64] Jia Deng, W. Dong, R. Socher, L. Li, K. Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR 2009*, 2009.
[65] Olga Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Zhiheng Huang, A. Karpathy, A. Khosla, Michael S. Bernstein, A. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115:211–252, 2015.
[66] Todd Holloway. Introduction to ensemble learning - featuring successes in the netflix prize competition, 2007.
[67] Neurips 2020 nlc2cmd, 2020.
[68] Samsung nlc2cmd, 2020.
[69] Nokia nlc2cmd submission hubris, 2020.
[70] Jetbrains nlc2cmd, 2020.
[71] Jeniya Tabassum, Mounica Maddela, W. Xu, and Alan Ritter. Code and named entity recognition in stackoverflow. *ArXiv*, abs/2005.01634, 2020.
[72] Jiatao Gu, Z. Lu, Hang Li, and V. Li. Incorporating copying mechanism in sequence-to-sequence learning. *ArXiv*, abs/1603.06393, 2016.
[73] Tao Yu, Rui Zhang, H. Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Y. Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, T. Chen, Alexander R. Fabbri, Z. Li, Luyao Chen, Y. Zhang, Shreya Dixit, V. Zhang, Caiming Xiong, R. Socher, Walter S. Lasecki, and Dragomir R. Radev. Cosql: A conversational text-to-sql challenge towards cross-domain natural language interfaces to databases. In *EMNLP/IJCNLP*, 2019.
[74] J. Nielsen. Usability engineering. In *The Computer Science and Engineering Handbook*, 1997.
[75] Kishore Papineni, S. Roukos, T. Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *ACL*, 2002.
[76] Ngoc M. Tran, H. Tran, Son Nguyen, H. Nguyen, and T. Nguyen. Does bleu score work for code migration? *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 165–176, 2019.

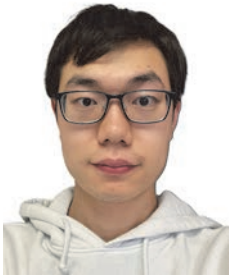
[77] Peter F. Brown, S. D. Pietra, V. D. Pietra, and R. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Comput. Linguistics*, 19:263–311, 1993.

[78] J. R. Medina and J. Kalita. Parallel attention mechanisms in neural machine translation. *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 547–552, 2018.

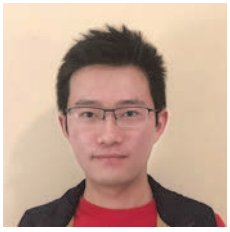
[79] Benyamin Ahmadnia, Parisa Kordjamshidi, and Gholamreza Haffari. Neural machine translation advised by statistical machine translation: The case of farsi-spanish bilingually low-resource scenario. *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1209–1213, 2018.

[80] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, 2017.

Biographies



Quchen Fu is a Ph.D. student at Vanderbilt University major in Computer Science, his research interest is NLP and Deep Learning. He got his Master's degree in CMU and he was TA for multiple courses including Cloud Computing and Cybersecurity. He interned at multiple companies including Tencent, Intel, and Microsoft. He is now a research assistant in Magnum research group under Dr. Jules White.



Zhongwei Teng is pursuing a Ph.D. in Computer Science in Vanderbilt University. His research interests include speech verification, NLP and machine learning.

Marco Georgaklis recently graduated from Vanderbilt University with a Bachelor's in Computer Science. He will be starting as a Software Engineer at Google in the Fall of 2022.

Jules White is Associate Dean of Strategic Learning Programs in the School of Engineering and Associate Professor of Computer Science in the Dept. of Computer Science at Vanderbilt University. He is a National Science Foundation CAREER Award recipient. His research has won multiple Best Paper Awards. He has also published over 150 papers. Dr. White's research

focuses on cyber-security and mobile/cloud computing in domains ranging from healthcare to manufacturing. His research has been licensed and transitioned to industry, where it won an Innovation Award at CES 2013, attended by over 150,000 people, was a finalist for the Technical Achievement at Award at SXSW Interactive, and was a top 3 for mobile in the Accelerator Awards at SXSW 2013. He has raised over $12 million in venture backing for his startup companies. His research is conducted through the Mobile Application computinG, optimizatoN, and secUrity Methods (MAGNUM) Group at Vanderbilt University, which he directs.

Douglas C Schmidt is the Cornelius Vanderbilt Professor of Computer Science, Associate Chair of Computer Science, Co-Director at the Data Science Institute, and a Senior Researcher at the Institute for Software Integrated Systems, all at Vanderbilt University. His research covers a range of software-related topics, including patterns, optimization techniques, and empirical analyses of frameworks and model-driven engineering tools that facilitate the development of mission-critical middleware for distributed real-time embedded (DRE) systems and intelligent mobile cloud computing applications. Dr. Schmidt received B.A. and M.A. degrees in Sociology from the College of William and Mary in Williamsburg, Virginia, and an M.S. and a Ph.D. in Computer Science from the University of California, Irvine (UCI) in 1984, 1986, 1990, and 1994, respectively.