2022 International Conference on Software Engineering
Pittsburgh, PA, May 25th 2022

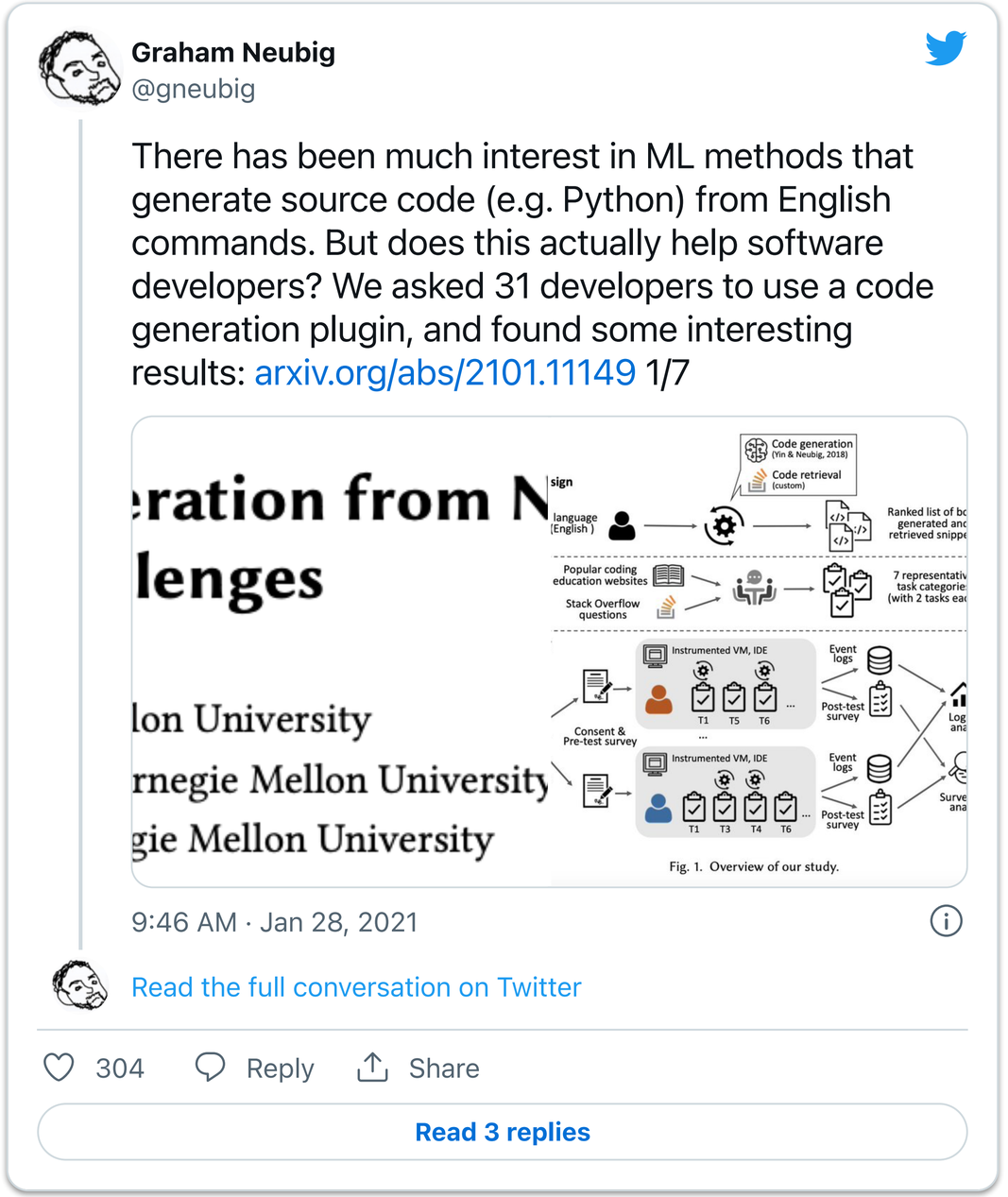
[Journal First - ACM TOSEM]

In-IDE Code Generation from Natural Language: Promise and Challenges

Frank Xu
@frankxu2004

Bogdan Vasilescu
@b_vasilescu

Graham Neubig
@gneubig



"I suspect that machines to be programmed in our native tongues —be it Dutch, English, American, French, German, or Swahili— are **as damned difficult to make as they would be to use**."

Edsger W. Dijkstra. 1979. On the foolishness of "natural language programming." In Program Construction. Springer, 51–53.

Dijkstra certainly got the “difficult to make” part right …

It took ~40 years

[EMNLP 2018]

TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation

Pengcheng Yin, Graham Neubig
Language Technologies Institute
Carnegie Mellon University
{pcyin,gneubig}@cs.cmu.edu

Abstract

We present TRANX, a transition-based neural semantic parser that maps natural language (NL) utterances into formal meaning representations (MRs). TRANX uses a transition system based on the *abstract syntax description language* for the target MR, which gives it two major advantages: (1) it is highly accurate, using information from the syntax of the target MR to constrain the output space and model the information flow, and (2) it is highly generalizable, and can easily be applied to new types of MR by just writing a new abstract syntax description corresponding to the allowable structures in the MR. Experiments on four different semantic parsing and code generation tasks show that our system is generalizable, extensible, and effective, registering strong results compared to existing neural semantic parsers.[1]

1 Introduction

Semantic parsing is the task of transducing natural language (NL) utterances into formal meaning representations (MRs). The target MRs can be defined according to a wide variety of formalisms for MRs, the design of semantic parsers, particularly neural network-based ones has generally focused on a small subset of tasks — in order to ensure the syntactic well-formedness of generated MRs, a parser is usually specifically designed to reflect the domain-dependent grammar of MRs in the structure of the model (Zhong et al., 2017; Xu et al., 2017). To alleviate this issue, there have been recent efforts in neural semantic parsing with general-purpose grammar models (Xiao et al., 2016; Dong and Lapata, 2018). Yin and Neubig (2017) put forward a neural sequence-to-sequence model that generates tree-structured MRs using a series of tree-construction actions, guided by the task-specific context free grammar provided to the model *a priori*. Rabinovich et al. (2017) propose the abstract syntax networks (ASNs), where domain-specific MRs are represented by abstract syntax trees (ASTs, Fig. 2 *Left*) specified under the abstract syntax description language (ASDL) framework (Wang et al., 1997). An ASN employs a modular architecture, generating an AST using specifically designed neural networks for each

[ACL 2020]

Incorporating External Knowledge through Pre-training for Natural Language to Code Generation

Frank F. Xu*, Zhengbao Jiang*, Pengcheng Yin, Bogdan Vasilescu, Graham Neubig
Carnegie Mellon University
{fangzhex,zhengbaj,pcyin,vasilescu,gneubig}@cs.cmu.edu

Abstract

Open-domain code generation aims to generate code in a general-purpose programming language (such as Python) from natural language (NL) intents. Motivated by the intuition that developers usually retrieve resources on the web when writing code, we explore the effectiveness of incorporating two varieties of external knowledge into NL-to-code generation: automatically mined NL-code pairs from the online programming QA forum StackOverflow and programming language API documentation. Our evaluations show that combining the two sources with data augmentation and retrieval-based data re-sampling improves the current state-of-the-art by up to 2.2% absolute BLEU score on the code generation testbed CoNaLa. The code and resources are available at https://github.com/neulab/external-knowledge-codegen.

1 Introduction

Semantic parsing, the task of generating machine executable meaning representations from natural

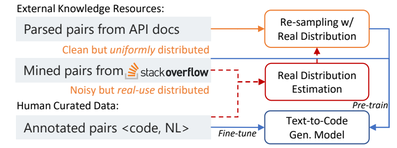


Figure 1: Our approach: incorporating external knowledge by data re-sampling, pre-training and fine-tuning.

2018; Iyer et al., 2018; Yin and Neubig, 2019) used a variety of models, especially neural architectures, to achieve good performance.

However, open-domain code generation for general-purpose languages like Python is challenging. For example, given the intent to *choose a random file from the directory contents of the C drive, ‘C:\\’*, one would expect the Python code snippet `random.choice(os.listdir('C:\\'))`, that realizes the given intent. This would involve not just generating syntactically correct code, but also using (and potentially combining) calls to APIs and libraries that implement some of the desired functionality. As we show in § 3, current code generation models still have difficulty generating the

Dijkstra certainly got the “difficult to make” part right …

It took ~40 years

Remove the first column of dataframe *df*

Code generation
(Yin & Neubig, 2018)

```
df.drop(df.columns[[0]])
```

Code retrieval

```
del df['column_name']
```

(Training set example is somewhat different still)

```
df = df.drop(df.columns[[0]], axis=1)
```

What about usability?

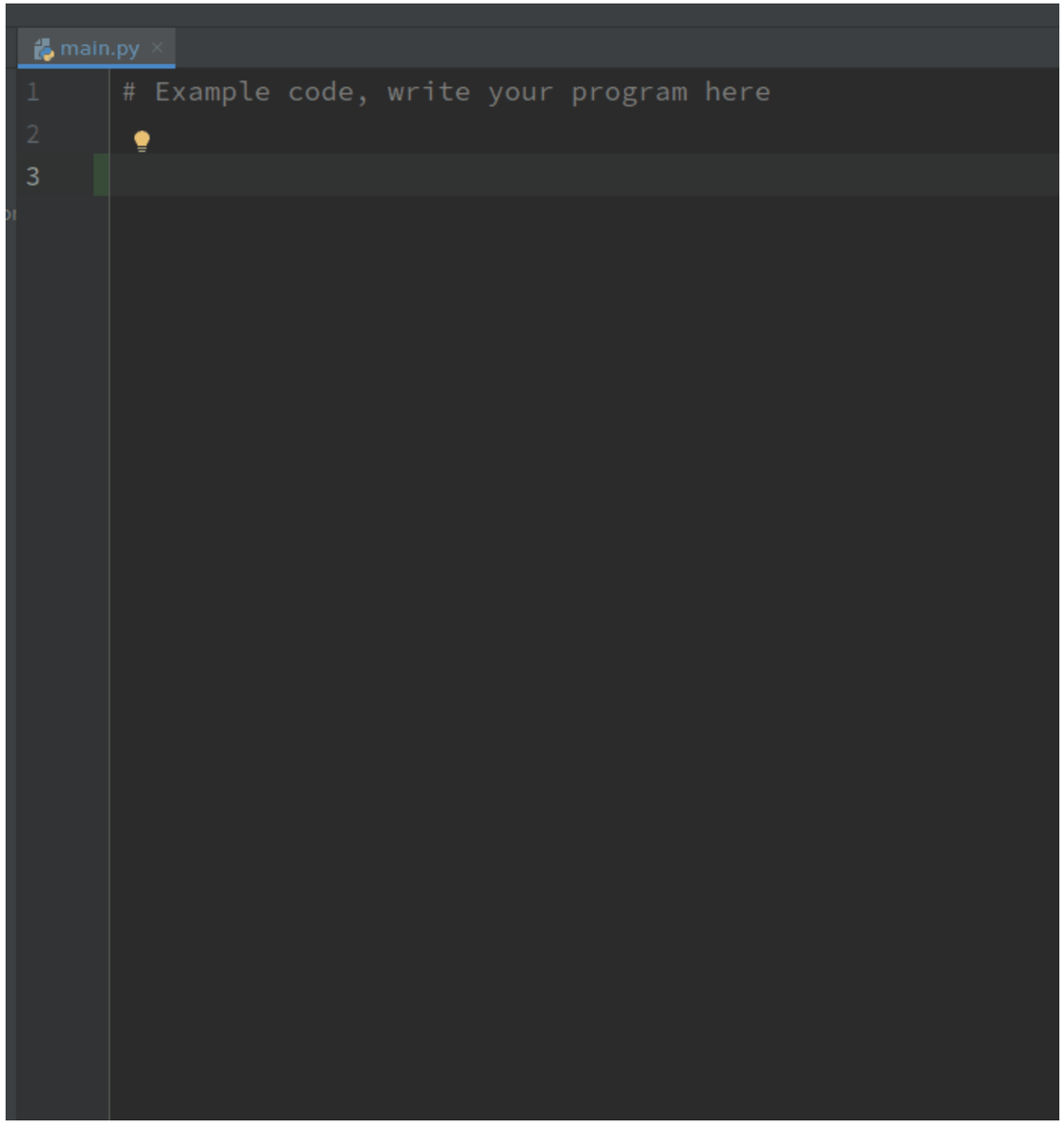
BLEU score X on some benchmark

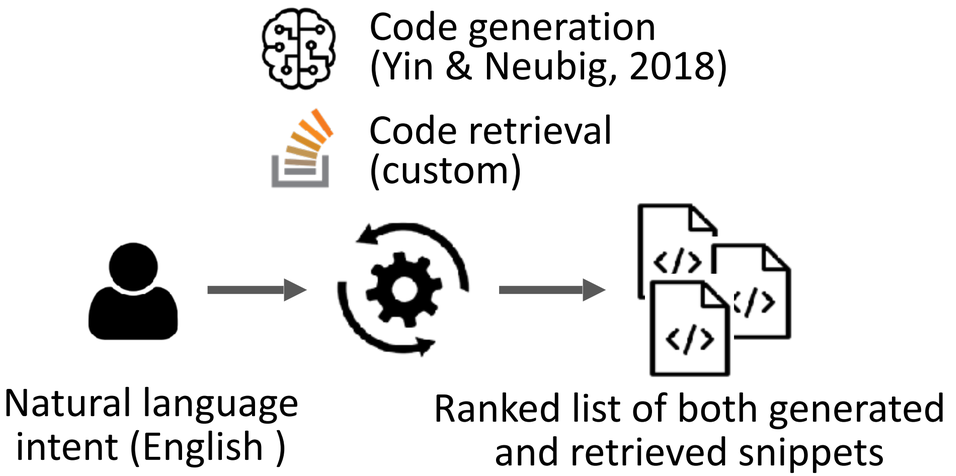
vs.

Usefulness in real-world scenarios

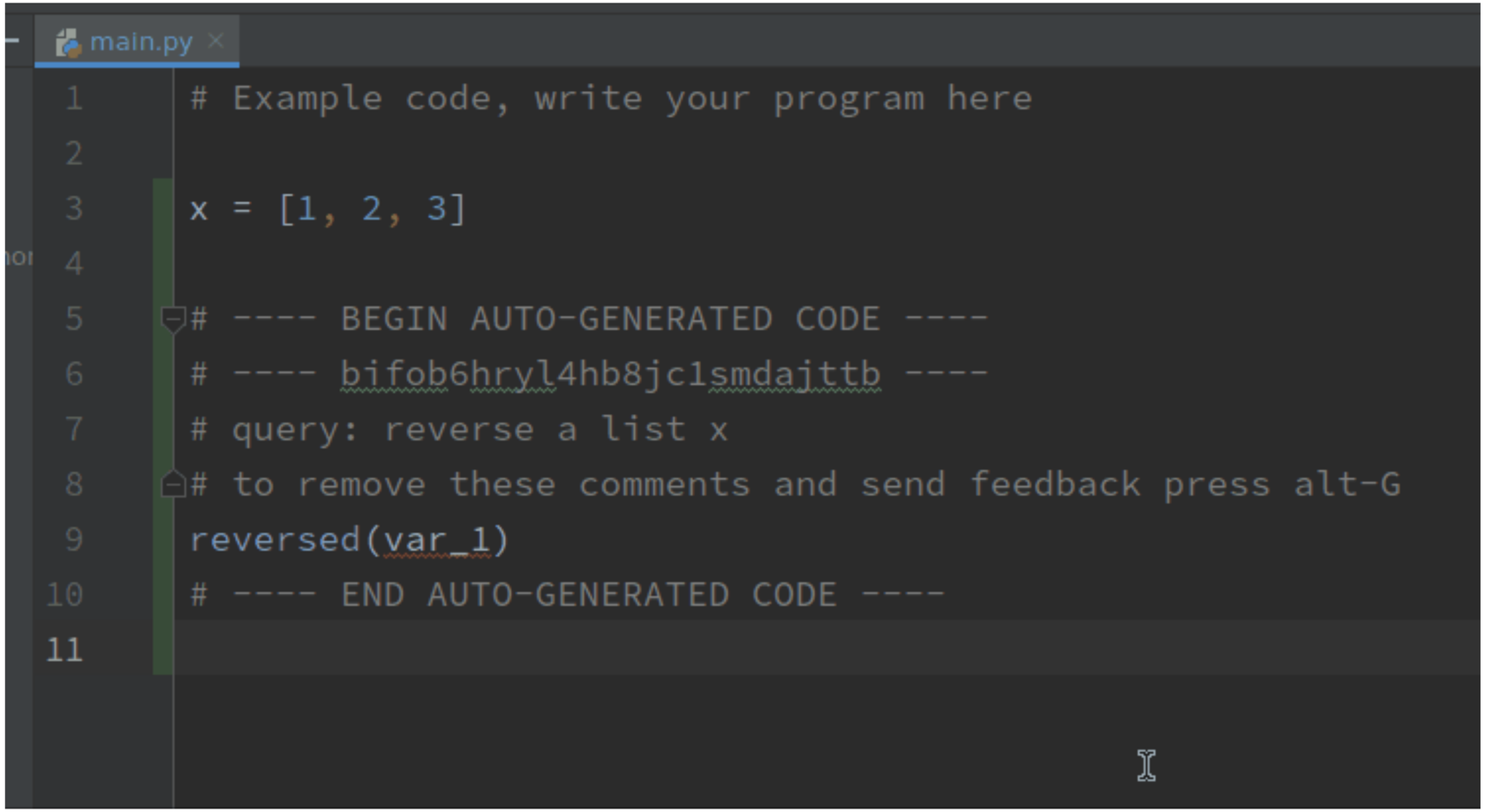
Our human study

Step 1: Develop an instrumented IDE plugin



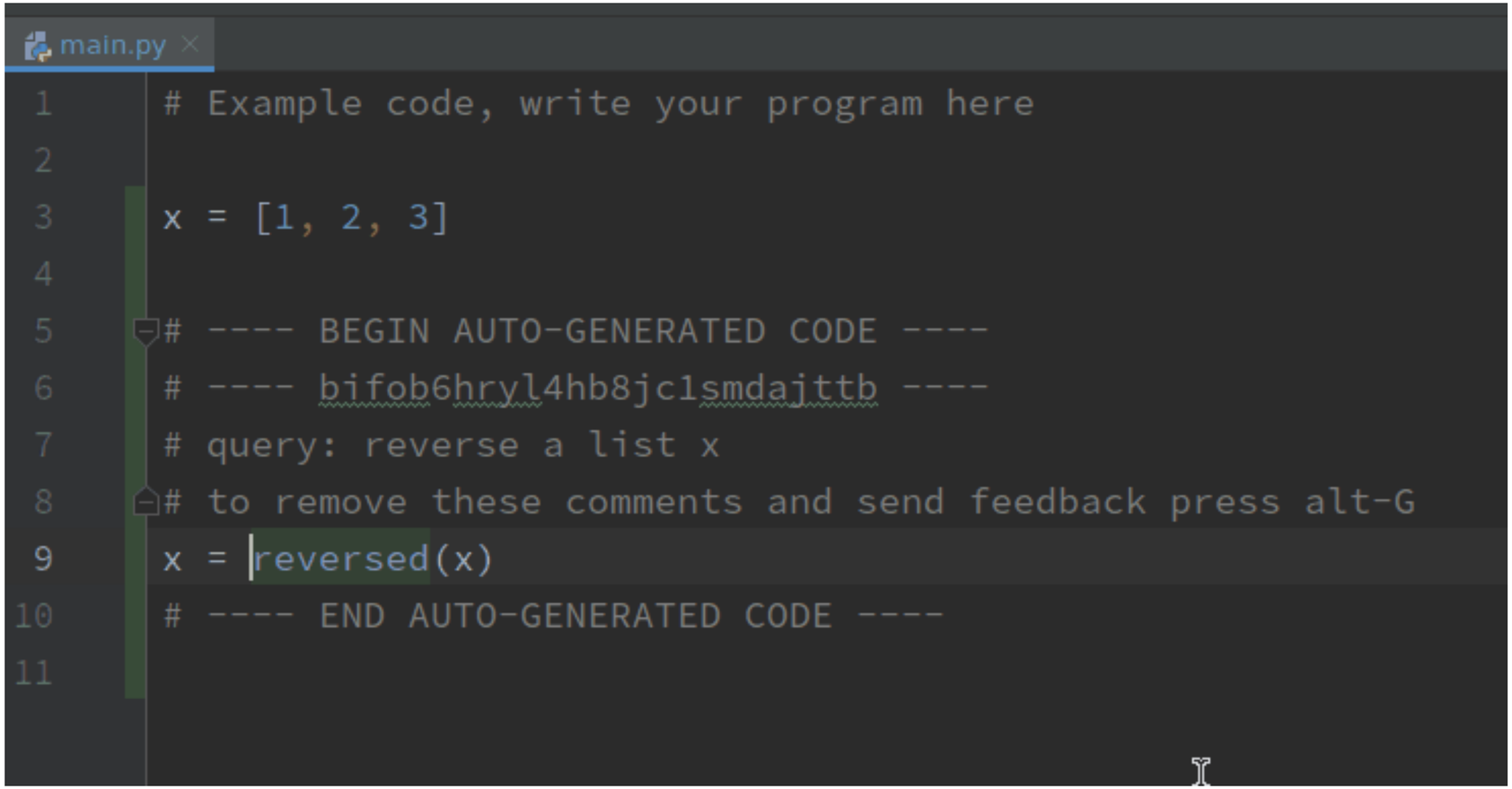


Step 1: Develop an instrumented IDE plugin



Telemetry

Step 1: Develop an instrumented IDE plugin

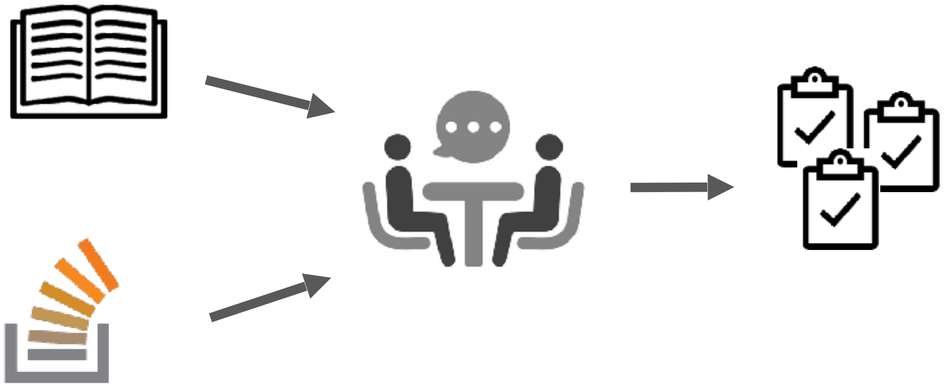


Telemetry

Step 2: Compile a set of realistic tasks

Popular coding education websites

Stack Overflow questions

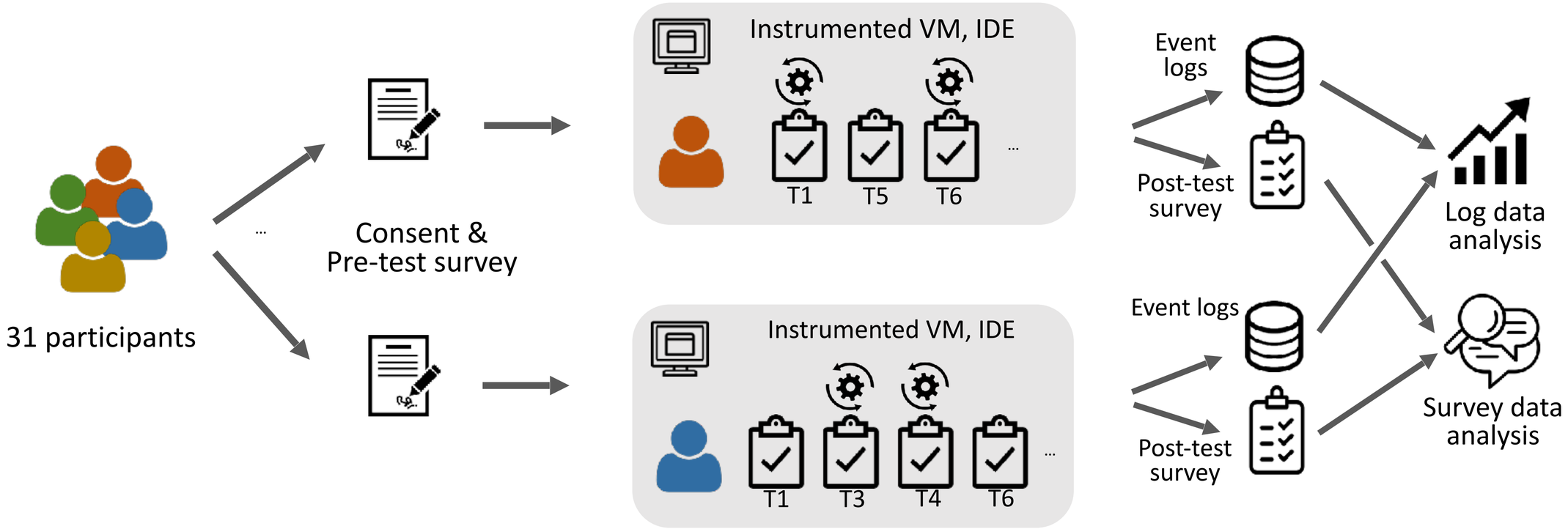


7 representative task categories (with 2 tasks each)

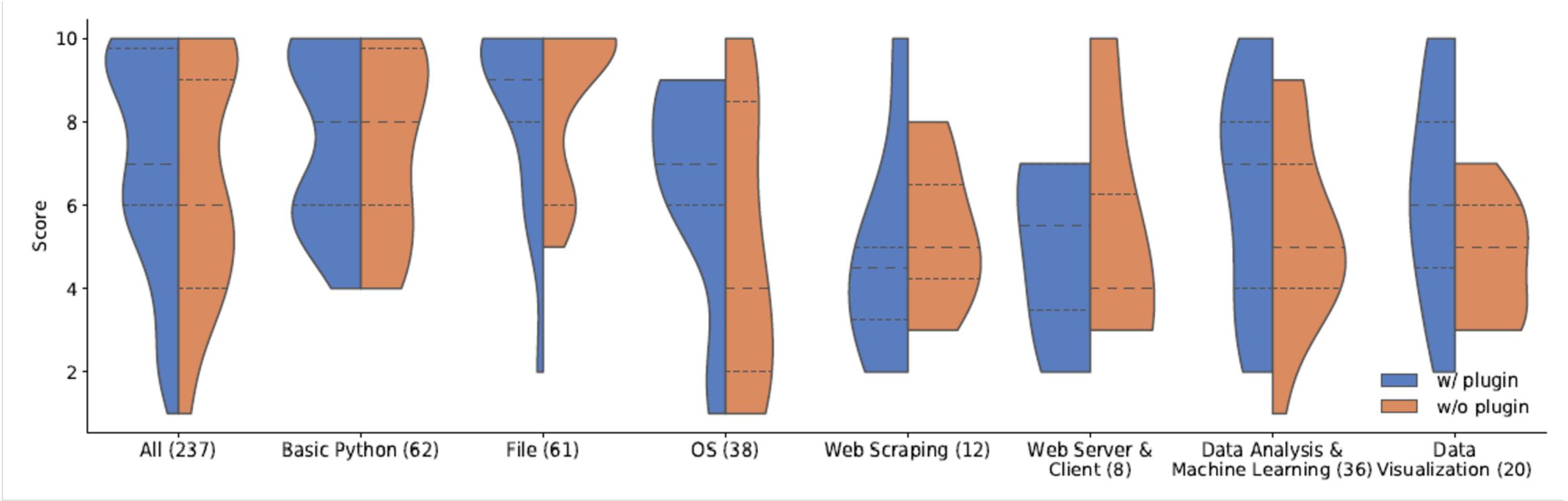
Step 2: Compile a set of realistic tasks

Category	*Tasks*	
Basic Python	T1-1	Randomly generate and sort numbers and characters with dictionary
	T1-2	Date & time format parsing and calculation with timezone
File	T2-1	Read, manipulate, and output CSV files
	T2-2	Text processing about encoding, newline styles, and whitespaces
OS	T3-1	File and directory copying, name editing
	T3-2	File system information aggregation
Web Scraping	T4-1	Parse URLs and specific text chunks from web page
	T4-2	Extract table data and images from Wikipedia page
Web Server & Client	T5-1	Implement an HTTP server for querying and validating data
	T5-2	Implement an HTTP client interacting with given blog post APIs
Data Analysis & ML	T6-1	Data analysis on automobile data of performance metrics and prices
	T6-2	Train and evaluate a multi-class logistic regression model given dataset
Data Visualization	T7-1	Produce a scatter plot given specification and dataset
	T7-2	Draw a figure with three grouped bar chart subplots aggregated from dataset

Step 3: 31 participants completing tasks



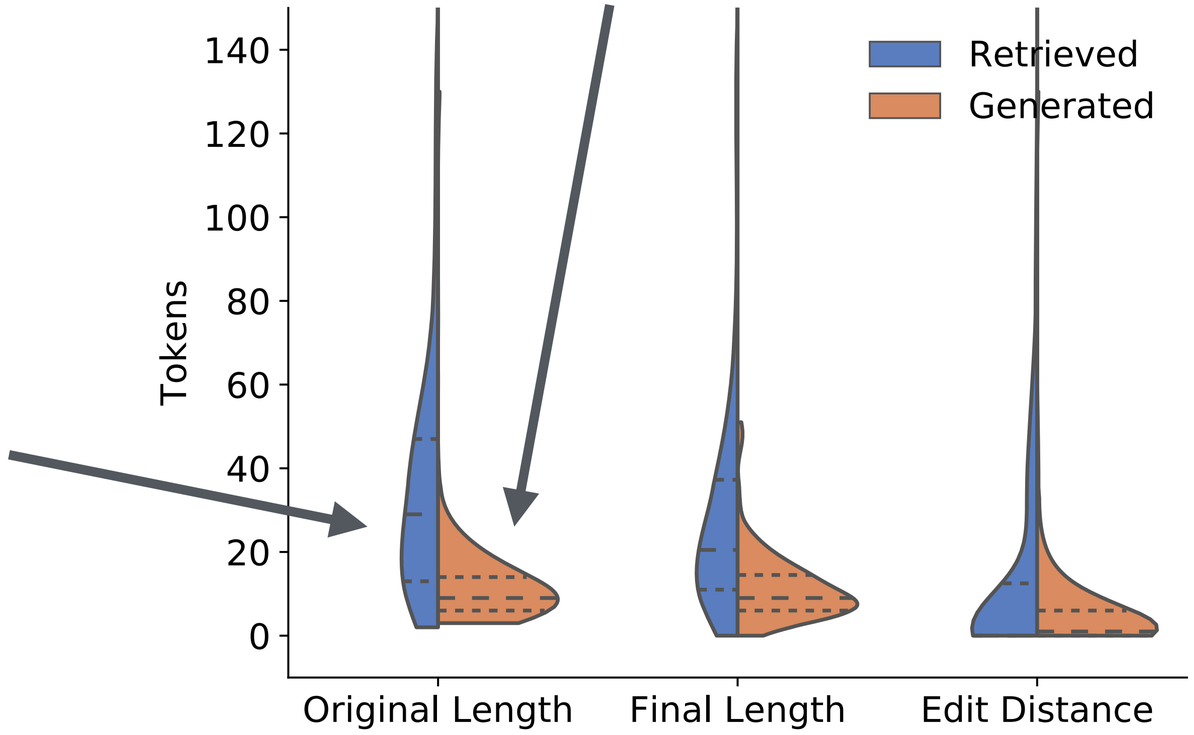
Highlight: No significant improvement in quantitative measures of speed, code quality, or program correctness when using the plugin



Highlight: Code generation and retrieval are useful in different settings.

Code generation methods work better for simpler queries on basic Python functionality

Code retrieval is preferred for more complicated functionality

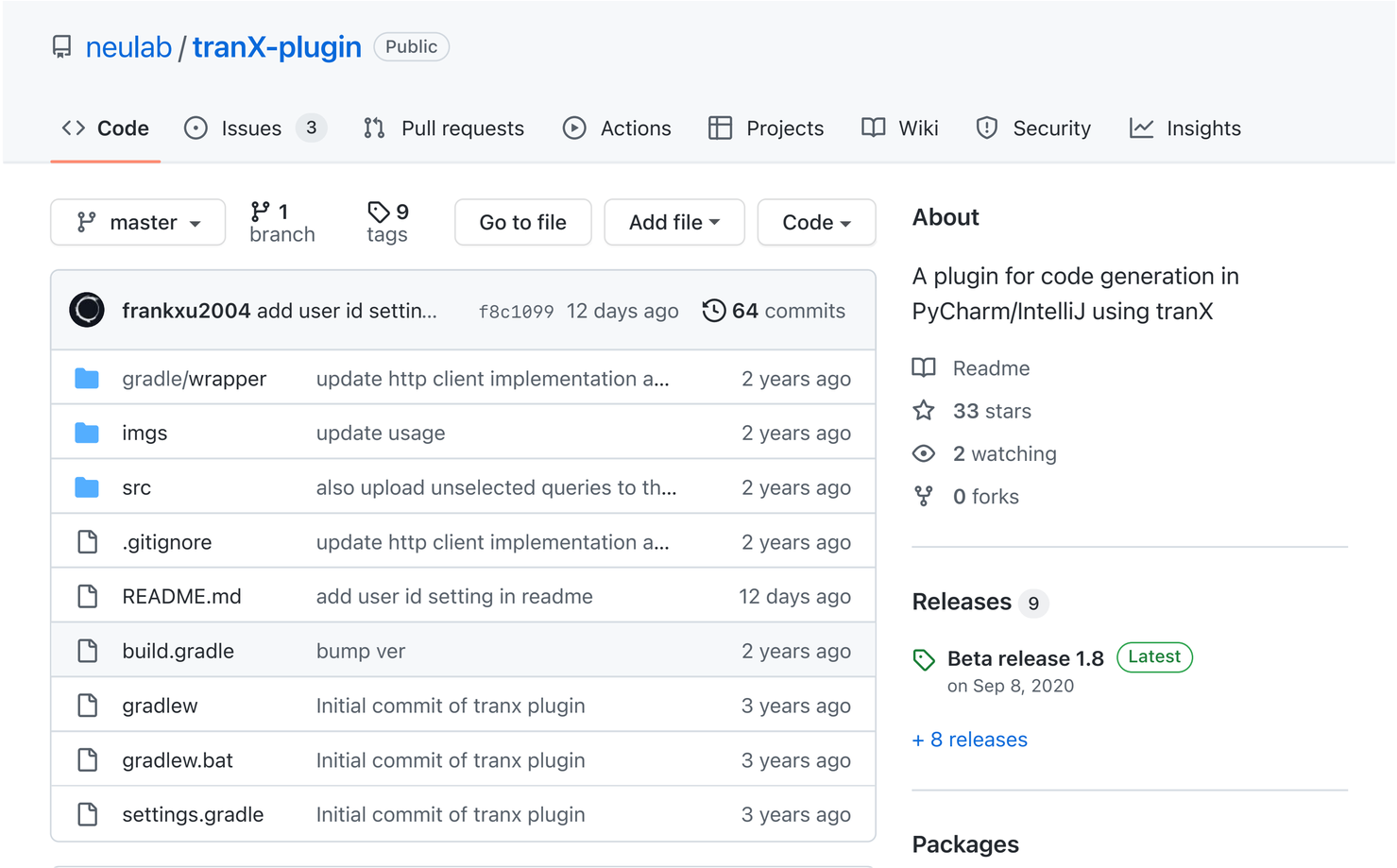


Highlight: In general, users liked using the plugin!

But we have a loooong way to go to make code generation systems practical

- *Combine code generation with code retrieval.* Our results suggest that some queries may be better answered through code retrieval techniques, and others through code generation. We recommend that future research continue to explore these types of approaches jointly, e.g., using hybrid models [40, 41] that may be able to combine the best of both worlds.
- *Consider the user's local context as part of the input.* Our oracle comparison revealed that users' natural language queries can often be disambiguated by considering the local context provided by the source files they were working in at the time, which in turn could lead to better performance of the code generation model. There is already convincing evidence from prior work that considering a user's local context provides unique information about what code they might type next [111]. In addition, some work on code retrieval has also considered how to incorporate context to improve retrieval results [17]; this may be similarly incorporated.
- *Consider the user's local context as part of the output.* Considering where in their local IDE users are when invoking an NL2Code assistant can also help with localizing the returned code snippets for that context. Some transformations are relatively simple, e.g., pretty printing and indentation. Other transformations may require more advanced program analysis but are still well within reach of current technology, e.g., renaming variables used in the returned snippet to match the local context (the Bing Developer Assistant code retrieval engine [115] already does this), or applying coding conventions [2].
- *Provide more context for each returned snippet.* Our study shows that NL2Code generation or retrieval systems can be useful when users already know what the right answer is, but they need help retrieving it. At the same time, many of our study participants reported lacking sufficient background knowledge, be it domain-specific or API-specific, to recognize when a plugin-returned code snippet is the right one given their query, or what the snippet does in detail. Future research should consider incorporating more context and documentation together with the plugin's results, which allows users to better understand the code, e.g., links to Stack Overflow, official documentation pages, explanations of domain-specific concepts, other API usage examples. One example of this is the work of Moreno et al. [78], which retrieves usage examples that show how to use a specific method.
- *Provide a unified and intuitive query syntax.* We observed that users are not always formulating queries in the way that we would expect, perhaps because they are used to traditional search engines that are more robust to noisy inputs and designed for keyword-based search. The NL2Code generation model we experimented with in this study was trained on natural language queries that are not only complete English sentences, but also include references to variables or literals involved with an intent, specially delimited by dedicated syntax (grave accents). As our respondents commented in the post-test survey, getting used to formulating queries this way takes some practice. Future research should consider not only what is the most natural way for users to describe their intent using natural language, but also how to provide a unified query syntax for both code generation and code retrieval, to minimize confusion. Robust semantic parsing techniques [8, 95] may also help with interpreting ill-specified user queries.
- *Provide dialogue-based query capability.* Dialogue-based querying could allow users to refine their natural language intents until they are sufficiently precise for the underlying models to confidently provide some results. Future systems may reference work on query reformulation in information retrieval, where the user queries are refined to improve retrieval results both for standard information retrieval [7] and code retrieval [39, 45]. In addition, in the NLP community there have been notable advancements recently in interactive semantic parsing [51, 119], i.e., soliciting user input when dealing with missing information or ambiguity while processing the initial natural language query, which could be of use as well.
- *Consider new paradigms of evaluation for code generation and retrieval systems.* Usage log data, such as the ones we collected here, is arguably very informative and useful for researchers looking to evaluate NL2Code systems. However, compared to automated metrics such as BLEU, such data is much less readily available. We argue that such data is worth collecting even if only in small quantities. For example, with little but high-quality data, one could still train a reranker [125] to try to select the outputs that a human user selected; if the predictive power exceeds that of BLEU alone, then the trained reranker could be used to automatically evaluate the quality of the generated or retrieved code more realistically than by using BLEU.

Conclusion: We need more human-centered approaches to evaluate ML-based tools within the software development workflow



https://github.com/neulab/tranX-plugin