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

FRANK F. XU, BOGDAN VASILESCU, and GRAHAM NEUBIG, Carnegie Mellon University

A great part of software development involves conceptualizing or communicating the underlying procedures and logic that needs to be expressed in programs. One major difficulty of programming is turning *concept* into *code*, especially when dealing with the APIs of unfamiliar libraries. Recently, there has been a proliferation of machine learning methods for code generation and retrieval from *natural language queries*, but these have primarily been evaluated purely based on retrieval accuracy or overlap of generated code with developer-written code, and the actual effect of these methods on the developer workflow is surprisingly unattested. In this article, we perform the first comprehensive investigation of the promise and challenges of using such technology inside the PyCharm IDE, asking, "At the current state of technology does it improve developer productivity or accuracy, how does it affect the developer experience, and what are the remaining gaps and challenges?" To facilitate the study, we first develop a plugin for the PyCharm IDE that implements a hybrid of code generation and code retrieval functionality, and we orchestrate virtual environments to enable collection of many user events (e.g., web browsing, keystrokes, fine-grained code edits). We ask developers with various backgrounds to complete 7 varieties of 14 Python programming tasks ranging from basic file manipulation to machine learning or data visualization, with or without the help of the plugin. While qualitative surveys of developer experience are largely positive, quantitative results with regards to increased productivity, code quality, or program correctness are inconclusive. Further analysis identifies several pain points that could improve the effectiveness of future machine learning-based code generation/retrieval developer assistants and demonstrates when developers prefer code generation over code retrieval and vice versa. We release all data and software to pave the road for future empirical studies on this topic, as well as development of better code generation models.

CCS Concepts: • **Software and its engineering → Software notations and tools**; **Automatic programming**; • **Human-centered computing → Natural language interfaces;**

Additional Key Words and Phrases: Natural language programming assistant, code generation, code retrieval, empirical study

ACM Reference format:
Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 29 (February 2022), 47 pages.
https://doi.org/10.1145/3487569

This research was supported by NSF Award No. 1815287 "Open-domain, Data-driven Code Synthesis from Natural Language."

Authors' address: F. F. Xu, B. Vasilescu, and G. Neubig, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, 15213; emails: {fangzhex, vasilescu, gneubig}@cs.cmu.edu.

1049-331X/2022/02-ART29 $15.00
https://doi.org/10.1145/3487569

1 INTRODUCTION

One of the major hurdles to programming is the time it takes to turn ideas into code [77]. All programmers, especially beginners but even experts, frequently reach points in a program where they understand conceptually what must be done next, but do not know how to create a concrete implementation of their idea or would rather not have to type it in if they can avoid it. The popularity of the Stack Overflow Q&A website is a great example of this need. Indeed, developers ask questions about how to transform ideas into code all the time, e.g., "How do I check whether a file exists without exceptions?,"[1] "How can I merge two Python dictionaries in a single expression?,"[2] and so on. Moreover, this need is likely to continue in the future, as new APIs appear continuously, and existing APIs change in non-backwards compatible ways [80], requiring recurring learning effort [57, 84].

Despite early skepticism towards the idea of "natural language programming" [26], researchers now widely agree on a range of scenarios where it can be useful to be able to formulate instructions using natural language and have the corresponding source code snippets automatically produced. For example, software developers can save keystrokes or avoid writing dull pieces of code [32, 86, 99, 115]; and non-programmers and practitioners in other fields, who require computation in their daily work, can get help with creating data manipulation scripts [38, 62].

Given a natural language query carrying the intent of a desired step in a program, there are two main classes of methods to obtain code implementing this intent, corresponding to two major research thrusts in this area. On the one hand, *code retrieval* techniques aim to search for and retrieve an existing code fragment in a code base; given the abundance of code snippets online, on platforms such as Stack Overflow, it is plausible that a lot of the code that one might write, especially for lower-level functionality and API usage primitives, already exists somewhere, therefore the main challenge is search. On the other hand, *code generation* techniques aim to synthesize code fragments given natural language descriptions of intent. This is typically a harder challenge than retrieval and therefore more ambitious, but it may be particularly useful in practice if those exact target code fragments do not exist anywhere yet and can be generated instead.

The early attempts at general-purpose code generation from natural language date back to the early to mid 2000s and resulted in groundbreaking but relatively constrained grammatical and template-based systems, e.g., converting English into Java [93] and Python [112]. Recent years have seen an increase in the scope and diversity of such programming assistance tools, as researchers have devised code generation techniques that promise to be more flexible and expressive using machine (deep) learning models trained on data from "Big Code" repositories such as GitHub and Stack Overflow; see Allamanis et al. [3] for an excellent survey of such techniques. Code retrieval systems have also improved dramatically in recent years, thanks to the increasing availability of source code online and more sophisticated information retrieval and machine learning techniques; perhaps the most popular current code retrieval system is Microsoft's Bing Developer Assistant [115], which is an adaptation of the Bing search engine for code.

While both types of methods (generation and retrieval) for producing appropriate code given natural language intents have received significant interest in machine learning circles, there is a surprising paucity of research using human-centered approaches [83] to evaluate the usefulness and impact of these methods *within the software development workflow*. An important open question is to what extent the typically high accuracy scores obtained during automatic evaluations on benchmark datasets will translate to real-world usage scenarios, involving software developers completing actual programming tasks. The former does not guarantee the latter. For example,

[1] https://stackoverflow.com/q/82831.
[2] https://stackoverflow.com/q/38987.

an empirical study on code migration by Tran et al. [110] showed that the BLEU [89] accuracy score commonly used in natural language machine translation has only weak correlation with the semantic correctness of the translated source code [110].

In this article, we take one step towards addressing this gap. We implemented two state-of-the-art systems for **natural language to code (NL2Code)** generation and retrieval as in-IDE developer assistants and carried out a controlled human study with 31 participants assigned to complete a range of Python programming tasks *with and without* the use of the two varieties of NL2Code assistance. Our results reveal that while participants in general enjoyed interacting with our IDE plugin and the two code generation and retrieval systems, surprisingly *there were no statistically significant gains in any measurable outcome when using the plugin.* That is, tasks with code fragments automatically generated or retrieved using our plugin were, on average, neither completed faster nor more correctly than tasks where participants did not use any NL2Code assistant. This indicates that despite impressive improvements in the intrinsic performance of code generation and retrieval models, there is a clear need to further improve the accuracy of code generation, and we may need to consider other extrinsic factors (such as providing documentation for the generated code) before such models can make sizable impact on the developer workflow.

In summary, the **main contributions** of this article are: (i) A hybrid code generation and code retrieval plugin for the Python PyCharm IDE, which takes as input natural language queries. (ii) A controlled user study with 31 participants observed across 7 types of programming tasks (14 concrete subtasks). (iii) An analysis of both quantitative and qualitative empirical data collected from the user study, revealing how developers interact with the NL2Code assistant and the assistant's impact on developer productivity and code quality. (iv) A comparison of code snippets produced by the two models, generation versus retrieval. (v) An anonymized dataset of events from our instrumented IDE and virtual environment, capturing multiple aspects of developers' activity during the programming tasks, including plugin queries and edits, web browsing activities, and code edits.

2 OVERVIEW OF OUR STUDY

The goal of our research is to elucidate to what extent and in what ways current natural language programming techniques for code generation and retrieval can be useful within the development workflow as NL2Code developer assistants. Our main interest is evaluating the usefulness in practice of state-of-the-art NL2Code *generation* systems, which have been receiving significant attention from researchers in recent years, but have so far only been evaluated on benchmark datasets using standard NLP metrics. However, as discussed above, code generation and code retrieval are closely related problems, with increasingly blurred lines between them; e.g., recent approaches to align natural language intents with their corresponding code snippets in Stack Overflow for retrieval purposes [122] use similar deep learning technology as some code generation techniques [123]. Therefore, it is important to also consider code retrieval systems when experimenting with and evaluating code generation systems.

Given this complementarity of the two tasks, we select as a representative example of state-of-the-art techniques for code generation the semantic parsing approach by Yin and Neubig [123]. In short, the approach is based on a tree-based neural network model that encodes natural language utterances and generates corresponding syntactically correct target code snippets; for example, the model can generate the Python code snippet "`x.sort(reverse=True)`" given the natural language input "sort list `x` in reverse order." We chose the approach by Yin and Neubig [123] over similar approaches such as those of Iyer et al. [49] and Agashe et al. [1], as it is the most general purpose and most naturally comparable to code retrieval approaches; see Section 9 for a discussion. For code retrieval, the closest analogue is Microsoft's proprietary Bing Developer Assistant [115], which takes English queries as input and returns existing matching code fragments from the Web

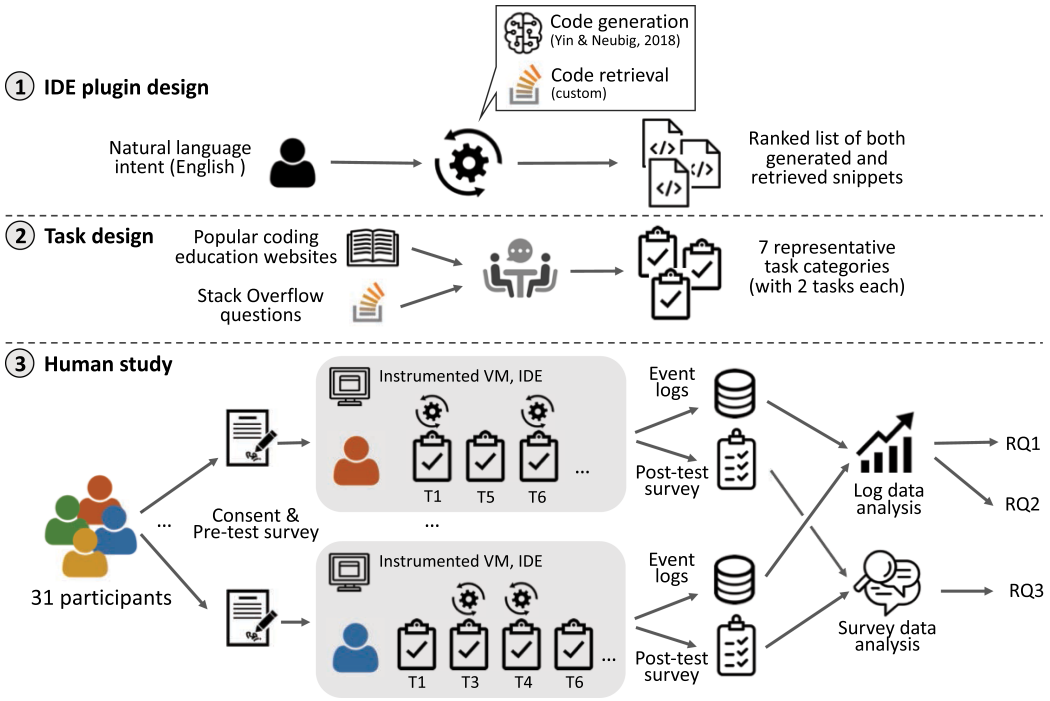


Fig. 1. Overview of our study.

using the Bing search engine. However, given the proprietary nature of this system, we build a custom Stack Overflow code search engine inspired by it rather than use the system itself.

We then designed and carried out the controlled human study summarized in Figure 1. First, we implement the two code generation and retrieval techniques as a custom plugin for the PyCharm[3] IDE, which takes as input natural language text intents and displays as output the corresponding code snippets generated and retrieved by the respective underlying models. Second, we compile 14 representative Python programming tasks across 7 task categories with varying difficulty, ranging from basic Python to data science topics. Third, we recruit 31 participants with diverse experience in programming in Python and with the different task application domains. Then, using an instrumented virtual environment and our IDE plugin, we collect quantitative and qualitative data about task performance and subjective tool use from each participant, as well as over 170 person hours of telemetry data from the instrumented environment.

Finally, we analyze these data to answer three research questions, as follows:

RQ$_1$. *How does using a NL2Code developer assistant affect task completion time and program correctness?* This research question investigates quantitative differences in outcome variables between tasks completed in the treatment and control conditions. To this end, we use the log data from our instrumented virtual environment to compute task completion times, and rubric-based manual scoring of the solutions submitted by study participants to evaluate program correctness. Then, we use multivariate mixed-effects regression modeling to analyze the data. We expect that using the plugin developers can complete tasks faster, without compromising solution quality.

[3]https://www.jetbrains.com/pycharm/.

RQ$_2$. *How do users query the NL2Code assistant, and how does that associate with their choice of generated vs. retrieved code?* This research question investigates quantitatively three dimensions of the inputs and outputs of the NL2Code plugin. Again using log data from our instrumented virtual environment, we first model how the natural language input queries differ when study participants favor the code snippets returned by the code generation model over those returned by the code retrieval model. Second, we evaluate the quality of the natural language queries input by study participants in terms of their ability to be answerable by an oracle (human expert), which is also important for the success of NL2Code systems in practice, in addition to the quality of the underlying code generation or retrieval systems. Third, we study how the length and the frequency of different types of tokens changes after study participants edit the candidate code snippets returned by the NL2Code plugin, which could indicate ways in which even the chosen code snippets are still insufficient to address the users' needs.

RQ$_3$. *How do users perceive the usefulness of the in-IDE NL2Code developer assistant?* Finally, this research question investigates qualitatively the experience of the study participants interacting with the NL2Code plugin and underlying code generation and retrieval models.

In the remainder of this article, Sections 3–4 describe our study setup in detail; then Sections 5–7 present our answers to the research questions; Section 8 discusses implications; and Section 9 discusses related work.

Following best practices for empirical software engineering research [107, 116], we make our study replicable, publishing our plugin prototype, instrumented virtual environment, data extraction and analysis scripts, and the obtained anonymized raw data; see the online appendices at https://github.com/neulab/tranX-plugin and https://github.com/neulab/tranX-study.

3 NL2CODE IDE PLUGIN DESIGN

We designed and built a joint NL2Code generation and retrieval plugin for PyCharm, a popular Python IDE. Our plugin is open source and available online.[4] As mentioned above, the plugin takes as input an English query describing the user's intent and gives as output a ranked list of the most relevant code snippets produced by each of the two underlying code generation and retrieval systems. Using IDE plugins to query Web resources such as Stack Overflow is expected to be less disruptive of developers' productivity than using an external Web browser, since it reduces context switching [9, 91]. Moreover, there exist already a number of IDE plugins for Web/Stack Overflow search and code retrieval [17, 91, 98, 115], therefore the human-computer interaction modality should feel at least somewhat natural to study participants.

The Underlying Code Generation System. For code generation, we use the model by Xu et al. [117] (available online[5]), which is an improved version of the tree-based semantic parsing model by Yin and Neubig [124], further pre-trained on official API documentation in addition to the original training on Stack Overflow questions and answers.[6]

This model reports state-of-the-art accuracy on the CoNaLa benchmark dataset [122], a benchmark dataset of intent/code pairs mined from Stack Overflow and standardly used to evaluate code generation models. Accuracy is computed using the BLEU score [89], a standard metric used in the NLP community, which measures the token-level overlap between the generated code and a reference implementation. As discussed above, the BLEU score (and similar automated metrics) are typically not sufficiently sensitive to small lexical differences in token sequence that can greatly

[4] At https://github.com/neulab/tranX-plugin.
[5] https://github.com/neulab/external-knowledge-codegen.
[6] We deployed the model on an internal research server and exposed a HTTP API that the plugin can access; queries are fast enough for the plugin to be usable in real time.

Table 1. Examples, where ✓ Is the Ground-truth Code Snippet, ♣ Is the Output from the State-of-the-Art Code Generation Model, and ♠ Is the First Candidate Retrieved from Stack Overflow Using Bing Search

Open a file "`f.txt`" in write mode.
✓ `f = open('f.txt', 'w')`
♣ `f = open('f.txt', 'w')`
♠ `with open(''users.txt'', ''a'')as f: f.write(username + ''\n'')`
Remove first column of dataframe *df*.
✓ `df = df.drop(df.columns[[0]], axis=1)`
♣ `df.drop(df.columns[[0]])`
♠ `del df['column_name']`
Lower a string *text* and remove non-alphanumeric characters aside from space.
✓ `re.sub(r'[^ \sa-zA-Z0-9]', '', text).lower().strip()`
♣ `re.sub(r'[^ \sa-zA-Z0-9]', '', text)`
♠ `re.sub(r'[^ \sa-zA-Z0-9]', '', text).lower().strip()`

alter the semantics of the code [110], hence our current human-centered study. Still, qualitatively, it appears that the model can generate reasonable code fragments given short text inputs, as shown in Table 1. Note how the model can generate syntactically correct code snippets by construction; demonstrates ability to identify and incorporate a wide variety of API calls; and also has the ability to copy important information such as string literals and variable names from the input natural language intent, in contrast to the code retrieval results. When displaying multiple generation results in the plugin described below, these results are ordered by the conditional probability of the generated code given the input command.

The Underlying Code Retrieval System. For code retrieval, similarly to a number of recent works on the subject [17, 91, 115], we implement a wrapper around a general-purpose search engine, specifically the Bing[7] search engine.[8] The wrapper queries this search engine for relevant questions on Stack Overflow,[9] the dominant programming Q&A community, and retrieves code from the retrieved pages. A dedicated search engine already incorporates advanced indexing and ranking mechanisms in its algorithms, driven by user interaction data, therefore it is preferable to using the internal Stack Overflow search engine directly [115].

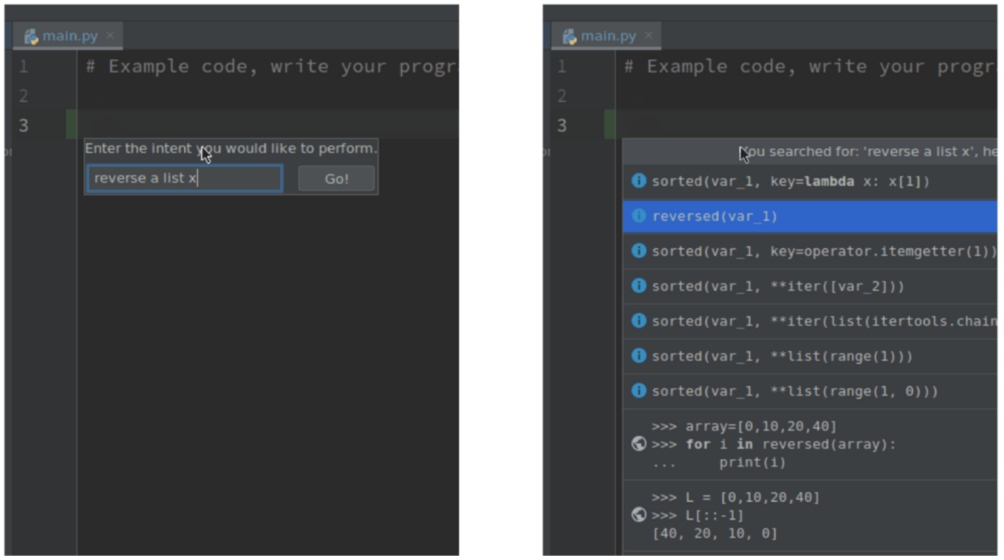
Specifically, we add the "Python" prefix to all user queries to confine the search to the Python programming language domain and add "site:stackoverflow.com" to confine the results to the Stack Overflow platform. We do not structurally alter the queries otherwise, e.g., we do not remove variables referenced therein, if any, although we do strip away grave accents that are part of the code generation model's syntax.[10] For the query example mentioned above, the actual query string

[7] https://www.bing.com/.

[8] We chose Bing rather than other alternatives such as Google due to the availability of an easily accessible search API.

[9] https://stackoverflow.com/.

[10] To mitigate concerns that user queries using the specified syntax (command form sentences and including variable names) may adversely affect the retrieval results, after the full study was complete, we modified 59 user-issued queries that were indeed complete sentences with full variable names, converting them into short phrases without variable names and reran the retrieval. We then compared the results and manually annotated the number of times the search engine returned a result that we judged was sufficient to understand how to perform the programming task specified by the user's intent. As a result, the user-written full intent resulted in a sufficient answer 34/59 times, and the simplified intent without variable names returned a sufficient answer 36/59 times, so it appears that including variable names has a marginal to no effect on whether the search engine was able to provide a good top-1 result. We also measured the exact-match overlap between the top-1 results and found it to be 22/59, and overlap between the top-7 result lists was 182/(59*7).



(a) Query input interface (b) Code snippet candidates

Fig. 2. Screenshots of the in-IDE plugin taking a natural language query as input and listing code snippet candidates from both code generation and code retrieval.

for Bing search would become "Python reverse a list x site:stackoverflow.com." For each Stack Overflow question page retrieved, we then extract the code snippets from the top three answers into a ranked list, sorted descending by upvotes. The code snippet extraction procedure follows Yin et al. [122] for identifying the code part of the answer, based on Stack Overflow-specific syntax highlighting and heuristics. When displaying multiple retrieval results, these results are ordered by the order they appeared in Bing search engine results, and the ordering of answers inside SO posts is done by upvotes.

Table 1 shows a few example outputs. Note how the retrieval results sometimes contain spurious code, not part of the natural language intent (first example), and otherwise seem to complement the generation results. Indeed, in the second example the generation result is arguably closer to the desired answer than the retrieval result, with the opposite situation in the third example.

Interacting with the Plugin. Figure 2 illustrates the plugin's user interface. The user first activates the query interface by pressing a keyboard shortcut when the cursor is in the IDE's editor. A popup appears at the current cursor position (Figure 2(a)), and the user can enter a command in natural language that they would like to be realized in code (e.g., "reverse a list `x`"[11]). The plugin then sends the request to the underlying code generation and code retrieval systems and displays a ranked list of results, with the top 7 code generation results at the top, followed by the top 7 code retrieval results (Figure 2(b)); 14 results are displayed in total.[12]

[11]Note the special syntax used to mark explicit variables; see Appendix F for full syntax details.

[12]We note that the main motivation for this ordering is that the generation results tend to be significantly more concise than the retrieval results (Figure 6). If we put the retrieval results first, then it is likely that the users would rarely scroll past the retrieval results and view the generation results due to issues of screen real-estate. It is important to consider that alternative orderings may result in different experimental results, although examining alternate orderings was not feasible within the scope of the current study.

(a) Generated code with errors in the context (b) The user fixes the error and uploads

Fig. 3. Screenshots of fixing the small errors in generated code and upload the correct snippet.

The number 7 was chosen subjectively, trying to maximize the amount and diversity of resulting code snippets while minimizing the necessary screen space to display them and, therefore, the amount of scrolling expected from study participants looking to inspect all the plugin-returned results. After completing the current study, we found that the most relevant code snippets are typically within the top 3 results, and thus a smaller number of candidates may be sufficient. While the number and ordering of candidates has the potential to have a significant impact on the efficiency and efficacy of the developer assistant, a formal evaluation of this impact is beyond the scope of this work.

If a code snippet is selected, then the code snippet is then inserted in the current cursor's position in the code editor. The user's selection is also recorded by our instrumentation in the back end. Understandably, some returned code snippets may not be directly suitable for the context inside the editor, so the user is welcome (and encouraged by the instructions we give as part of our human study) to edit the auto-inserted code snippets to fit their specific intent. After the edit is done, the user is asked to upload their edits to our server, along with the context of the code, using a dedicated key combination or the IDE's context menu. The process is illustrated in Figure 3. The edit data enable us to analyze how many and what kind of edits the users need to make to transform the auto-generated code to code that is useful in their context.[13]

4 HUMAN STUDY DESIGN

Given our NL2Code joint code generation and retrieval IDE plugin above, we designed and carried out a human study with 31 participants assigned to complete a range of Python programming tasks in both control (no plugin) and treatment (plugin) conditions.

4.1 Task Design

To emulate real world Python development activities, but also fit within the scope of a user study, we compiled a set of 14 reasonably sized Python programming tasks, organized into 7 categories (2 tasks per category) that span a diversity of levels of difficulty and application domains.

We started by identifying representative task categories that many users would encounter in practice. To that end, we analyzed two sources. First, we manually reviewed all the Python programming courses listed on three popular coding education websites (Udacity,[14] Codecademy,[15]

[13]The edit data may also be helpful as training data for improving code generation and retrieval models. We release our data publicly to encourage this direction in future work.
[14]https://www.udacity.com/courses/all.
[15]https://www.codecademy.com/catalog.

Table 2. Overview of Our 14 Python Programming 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

and Coursera[16]) to identify modules commonly taught across all websites that indicate common usage scenarios of the Python language. Second, we cross-checked if the previously identified use cases are well represented among frequently upvoted questions with the [python] tag on Stack Overflow, which would further indicate real programmer needs. By searching the category name, we found that each of our identified categories covers more than 300 questions with more than 10 upvotes on Stack Overflow. We iteratively discussed the emerging themes among the research team, refining or grouping as needed, until we arrived at a diverse but relatively small set of use cases, covering a wide range of skills a Python developer may need in practice.

In total, we identified seven categories of use cases, summarized in Table 2. For each of the 7 categories, we then designed two tasks covering use cases in the most highly upvoted questions on Stack Overflow. To this end, we searched Stack Overflow for the "python" keyword together with another keyword indicative of the task category (e.g., "python matplotlib," "python pandas"), selected only questions that were asking how to do something (i.e., excluding questions that ask about features of the language or about how to install packages), and drafted and iteratively refined after discussion among the research team tasks that would cover 3–5 of the most frequently upvoted questions.

We illustrate this process with the following example task for the "Data visualization" category[17]:

> By running `python3 main.py`, draw a scatter plot of the data in `shampoo.csv` and save it to `shampoo.png`. The plot size should be 10 inches wide and 6 inches high. The `Date` column is the x axis (some dates are missing from the data and in the plot the x axis should be completed with all missing dates without sales data). The date string shown on the plot should be in the format `(YYYY-MM-DD)`. The `Sales` column is the y axis. The graph should have the title "Shampoo Sales Trend." The font size of the title, axis labels, and x & y tick values should be 20pt, 16pt, and 12pt, respectively. The scatter points should be colored `purple`.

This task covers some of the top questions regarding data visualization with `matplotlib` found on Stack Overflow through the approach described above:

(1) How do you change the size of figures drawn with matplotlib?[18]

[16] https://www.coursera.org/.
[17] Corresponding to the search https://stackoverflow.com/search?tab=votes&q=python%20matplotlib.
[18] https://stackoverflow.com/questions/332289/how-do-you-change-the-size-of-figures-drawn-with-matplotlib.

(2) How to put the legend out of the plot?[19]
(3) Save plot to image file instead of displaying it using Matplotlib?[20]
(4) How do I set the figure title and axes labels font size in Matplotlib?[21]

For each task designed, we also provide the user with required input data or directory structure for their program to work on, as well as example outputs (console print-outs, output files & directories, etc.) so they could verify their programs during the user study.

Table 2 summarizes the 14 tasks. The full task descriptions and input/output examples can be found online, as part of our replication package at https://github.com/neulab/tranx-study. The tasks have varying difficulties, and on average each task would take about 15–40 minutes to complete.

4.2 Participant Recruitment & Task Assignments

Aiming to recruit participants with diverse technical backgrounds but at least some programming experience and familiarity with Python to be able to complete the tasks, we advertised our study in two ways: (1) inside the university community through personal contacts, mailing lists, and Slack channels, hoping to recruit researchers and students in computer science or related areas; (2) on the freelancer platform Upwork,[22] hoping to attract participants with software engineering and data science experience. We promised each participant US $5 per task as compensation; each participant was expected to complete multiple tasks.

To screen eligible applicants, we administered a pre-test survey to collect their self-reported levels of experience with Python and with each of the 7 specific task categories above; see Appendix B for the actual survey instrument. We only considered as eligible those applicants who reported at least some experience programming in Python, i.e., a score of 3 or higher given the answer range [1: very inexperienced] to [5: very experienced]; 64 applicants satisfied these criteria.

We then created personalized task assignments for each eligible applicant based on their self-reported levels of experience with the 7 specific task categories (see Appendix C for the distributions of participants' self reported experience across tasks), using the following protocol:

(1) To keep the study relatively short, we only assign participants to a total of 4 task categories (8 specific tasks, 2 per category) out of the 7 possible.
(2) Since almost everyone eligible for the study reported being at least somewhat experienced with the first 2 task categories (Basic Python and File), we assigned everyone to these 2 categories (4 specific tasks total). Moreover, we assigned these 2 categories first and second, respectively.
(3) For the remaining 5 task categories, sorted in increasing complexity order,[23] we rank them based on a participant's self-reported experience with that task genre, and then assign the participant to the top 2 task categories with most experience (another 4 specific tasks total).

Note that this filtering by experience is conducive to allowing participants to finish the tasks in a reasonable amount of time and reflective of a situation where a developer is working in their domain of expertise. However, at the same time it also means that different conclusions might be reached if novice programmers or programmers without domain expertise used the plugin instead.

Next, we randomly assigned the first task in a category to either the treatment condition, i.e., the NL2Code plugin is enabled in the virtual environment IDE and the participants are instructed

[19] https://stackoverflow.com/questions/4700614/how-to-put-the-legend-out-of-the-plot.
[20] https://stackoverflow.com/questions/9622163/save-plot-to-image-file-instead-of-displaying-it-using-matplotlib.
[21] https://stackoverflow.com/questions/12444716/how-do-i-set-the-figure-title-and-axes-labels-font-size-in-matplotlib.
[22] https://www.upwork.com/.
[23] The task identifiers in Table 2 reflect this order.

to use it,[24] or the control condition, i.e., the NL2Code plugin is disabled. The second task in the same category is then automatically assigned to the other condition, e.g., if the plugin is on for task1-1, then it should be off for task1-2. Therefore, each participant was asked to complete 4 tasks out of 8 total using the NL2Code plugin, and 4 without.

Finally, we invited all eligible applicants to read the detailed study instructions, access the virtual environment, and start working on their assigned tasks. Only 31 out of the 64 eligible applicants after the pre-test survey actually completed their assigned tasks.[25] Their backgrounds were relatively diverse; of the 31 participants, 12 (39%) were software engineers and 11 (35%) were computer science students, with the rest being researchers (2, 6%), and other occupations (6, 19%). Our results below are based on the data from these 31 participants.

4.3 Controlled Environment

Participants worked on their assigned tasks inside a custom instrumented online virtual environment, accessible remotely. Our virtual machine is preconfigured with the PyCharm Community Edition IDE[26] and the Firefox Web browser; and it has our NL2Code plugin either enabled or disabled inside the IDE, depending on the condition. See Appendix A for complete technical details.

In addition, the environment logs all of the user's interactions with the plugin in the PyCharm IDE, including queries, candidate selections, and edits; all of the user's fine-grained IDE editor activities; the user's Web search/browsing activities inside Firefox; all other keystrokes inside the VM; and the source code for each one of the user's completed tasks.

To get a sense of how the source code evolves, whenever the user does not make modifications to the code for at least 1.5 seconds, the plugin also automatically uploads the current snapshot of the code to our server. The intuition behind this heuristic is that after a user makes some type of meaningful edit, such as adding or modifying an argument, variable, or function, they usually pause for a short time before the next edit. This edit activity granularity can be more meaningful than keystroke/character level, and it is finer grained than intent level or commit level edits.

Given that it is identifiable, we record participants' contact information (only for compensation purposes) separately from their activity logs. This Human Subjects research protocol underwent review and was approved by the Carnegie Mellon University Institutional Review Board.

4.4 Data Collection

To answer our research questions (Section 2), we collect the following sets of data:

Task Performance Data (RQ_1). The first research question compares measurable properties of the tasks completed with and without the help of our NL2Code IDE plugin and its underlying code generation and code retrieval engines. One would expect that if such systems are useful in practice, then developers would be able to complete programming tasks faster without compromising on output quality. To investigate this, we measure two variables related to how well study participants completed their tasks and the quality of the code they produced:

- *Task Completion Time.* Since all activity inside the controlled virtual environment is logged, including all keystrokes and mouse movements, we calculate the time interval between when

[24]Despite these instructions, some participants did not use the plugin even when it was available and when instructed. We discovered this while analyzing the data collected from the study and filtered out 8 participants that did not use the plugin at all. They do not count towards the final sample of 31 participants we analyze data from, even though they completed tasks.

[25]Note that 4 of the 31 participants did not complete all 8 of their assigned tasks. We include their data from the tasks they completed and do not consider the tasks they did not finish.

[26]https://www.jetbrains.com/pycharm/download/.

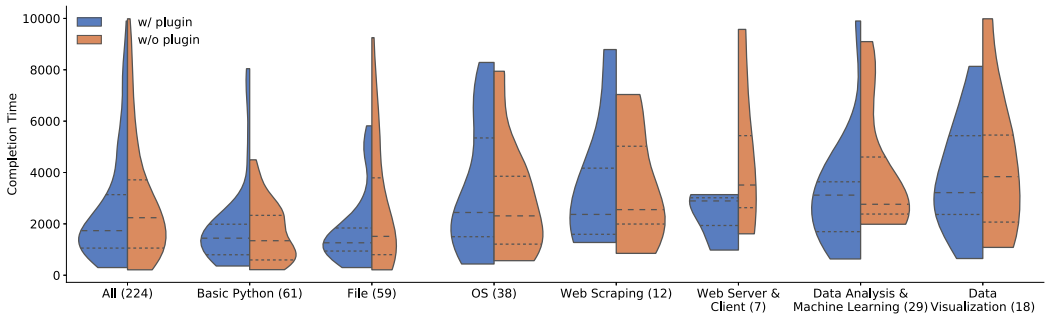


Fig. 4. Distributions of task completion times (in seconds) across tasks and conditions (w/ and w/o using the plugin). The horizontal dotted lines represent 25% and 75% quartiles, and the dashed lines represent medians.

a participant started working on a task (first keystroke inside the IDE) and when they uploaded their final submission to our server.

Recall that participants worked asynchronously and they may have decided to take breaks; we designed our virtual environment to account for this, with explicit pause/resume functionality. To account for possible breaks and obtain more accurate estimates of time spent on task, we further subtract the time intervals when participants used our explicit pause/resume functionality, as well as all intervals of idle time in which participants had no mouse or keyboard activity for two minutes or more (they may have taken a break without recording it explicitly).

Figure 4 shows the distributions of task completion times across the two conditions (with and without the plugin).

- *Task Correctness.* Following the common practice in computer science education [18, 25, 36], we design a rubric for each task concurrently with designing the task and later score each submission according to that rubric. We weigh all tasks equally, assigning a maximum score of 10 points to each. For each task, the rubric covers both basic aspects (e.g., runs without errors/exceptions; produces the same output as the example output provided in the task description) as well as implementation details regarding functional correctness (e.g., considers edge cases, implements all required functionality in the task description).

  For example, for the data visualization task described in Section 4.1, we created the following rubric, with the number in parentheses representing the point value of an item, for a total of 10 points: (i) Runs without errors (2); (ii) Correct image output format (png) (2); (iii) Read in the raw data file in correct data structure (1); (iv) Correct plot size (1); (v) Correctly handle missing data points (1); (vi) Date (x axis) label in correct format (1); (vii) Title set correctly (1); (viii) Font size and color set according to specification (1).

  To reduce subjectivity, we graded each submission blindly (i.e., not knowing whether it came from the control or treatment condition) and we automated rubric items when possible, e.g., using input-output test cases for the deterministic tasks and checking if the abstract syntax tree contains nodes corresponding to required types (data structures) such as dictionaries. See our online appendix[27] for the complete rubrics and test cases for all tasks.

  Figure 5 shows the distributions of scores across tasks, between the two conditions.

[27] https://github.com/neulab/tranx-study/blob/master/rubrics.md.

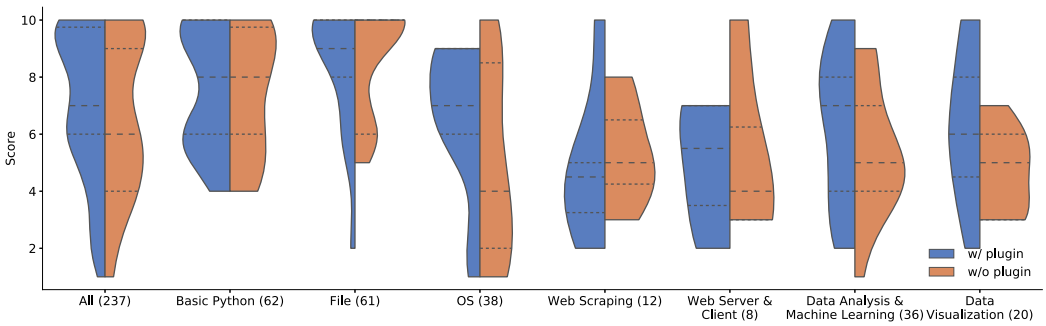


Fig. 5. Distributions of task correctness scores (0–10 scale) across tasks and conditions. The horizontal dotted lines represent 25% and 75% quartiles, and the dashed lines represent medians.

Plugin Queries, Snippets, and User Edits (RQ_2). We record user queries using the plugin, both the generated and retrieved code snippet candidates returned for the query, and the user selection from the candidates to insert into their source code. We use the data to analyze the NL queries and whether users preferred to use generated vs. retrieved code. In addition, we also record the user edits after inserting the code snippet from the plugin, along with the code context for the analysis on post edits required after using the plugin.

Participant Perceptions of Tool Use (RQ_3). We ran short post-test surveys after every task and a final post-test survey at the end of the study as a whole (see Appendix D for instruments) to collect data on the participants' subjective impressions of using the NL2code plugin and interacting with the code generation and code retrieval systems. We asked Likert-style and open-ended questions about aspects of using the plugin the participants enjoyed and aspects they wish to see improved.

Next, we describe how we analyzed these data and we answer each of our research questions.

5 RQ_1: NL2CODE PLUGIN EFFECTS ON TASK COMPLETION TIME AND PROGRAM CORRECTNESS

We start by describing our shared data analysis methodology, applied similarly to both variables corresponding to RQ_1, then present our results for each variable.

Methodology. Recall, we assign each participant a total of 8 tasks, 2 per task category, based on their experience levels with those categories; in each category, we randomly assign one of the 2 tasks to the NL2Code plugin (treatment) condition and the other task to the no plugin (control) condition. We then compute the three sets of outcome variables above.

The key idea behind our analysis is to compare the distributions of outcome variables between tasks completed in the treatment and control conditions. However, this comparison is not straightforward. First, our study design imposes a hierarchical structure during data collection, therefore the individual observations are not independent—by construction, the same participant will have completed multiple tasks over the course of the study. Moreover, tasks vary in difficulty, again by construction, therefore it is expected that their corresponding response variables, e.g., task completion times, can be correlated with the tasks themselves; e.g., on average, more complex tasks will take longer to complete. Finally, the participants vary in their self reported levels of Python and individual task category experience; we should separate experience-related effects from effects of using the plugin, if any.

Therefore, we use mixed-effects [34] as opposed to the more common fixed-effects regression models to analyze our data. Fixed-effects models assume that residuals are independently and

identically distributed, which is an invalid assumption in our case given the hierarchical nature of our data: E.g., responses for the different measurement occasions (tasks) within a given individual are likely correlated; a highly experienced Python programmer completing one task quickly is more likely to complete other tasks quickly as well. Mixed-effects models address this issue by having a residual term at each level, e.g., the observation level and the study participant level, in which case the individual participant-level residual is the so-called random effect. This partitions the unexplained residual variance into two components: higher-level variance between higher-level entities (study participants) and lower-level variance within these entities, between measurement occasions (tasks).

We consider two model specifications for each response variable. Our default model includes random effects for the individual and task, per the rationale above, a fixed effect for task category experience (e.g., participants with more machine learning experience should complete the machine learning task faster, on average), and a dummy variable to indicate the condition (plugin vs. no plugin). For example, for the task completion time response, we estimate the model:[28]

$$\texttt{completion\_time} = \texttt{experience} + \texttt{uses\_plugin} + (1|\texttt{user}) + (1|\texttt{task}). \quad (1)$$

As specified, our default model may suffer from heterogeneity bias [13]. Task category experience, a higher-level (i.e., individual-level as opposed to observation-level) predictor, varies both within and across study participants: Within participants, `experience` can vary across the 4 task categories—a user may be more experienced with basic Python than with data science; and across participants, `experience` with any given task category is likely to vary as well—some participants report higher experience with data science-related tasks than others. This means that `experience` (a fixed effect) and `user` (a random effect) may be "correlated." In turn, this may result in biased estimates, because both the within- and between-effect are captured in one estimate.

There are two sources of variation that can be used to explain changes in the outcome: (1) overall, more experienced programmers may be more efficient at completing tasks (group-level pattern); and (2) when becoming more experienced, programmers may also become more efficient at completing tasks (individual-level pattern). Therefore, to address potential heterogeneity bias, we split our fixed effect (`experience`) into two variables, each representing a different source of variation: a participant's average experience across all task categories (`experience_btw`) and the deviation for each task from the participants's overall mean experience (`experience_wi`). This process is known as de-meaning or person-mean centering [34]. This way, mixed-effects models can model both within- and between-subject effects [13], as recommended for a long time by Mundlak [79]. Taking the same task completion time response variable as an example (other variables are modeled analogously), our refined model becomes:

$$\texttt{completion\_time} = \texttt{experience\_btw} + \texttt{experience\_wi} + \texttt{uses\_plugin} + (1|\texttt{user}) + (1|\texttt{task}). \quad (2)$$

In both cases, the estimated coefficient for `uses_plugin` indicates the effect of using the plugin, *while holding fixed the effects of experience and other random user and task effects.*

For estimation, we used the functions `lmer` and `lmer.test` in R. We follow the traditional level for statistical significance when interpreting coefficient estimates, i.e., $p < 0.05$. As indicators of goodness of fit, we report a marginal (R^2_m) and a conditional (R^2_c) coefficient of determination for generalized mixed-effects models [50, 85], as implemented in the `MuMIn` package in R: R^2_m describes the proportion of variance explained by the fixed effects alone; R^2_c describes the proportion of variance explained by the fixed and random effects together.

Threats to Validity. Besides potential threats to statistical conclusion validity arising from the very nature of the data we are regressing over, discussed above and mitigated through our choice

[28] We are using the R syntax to specify random effects.

Table 3. LMER Task Performance Models (Default Specification)

	Dependent variable	
	Completion time	Correctness score
	(1)	(2)
Experience	−195.62	0.07
	(183.11)	(0.24)
Uses plugin	15.76	0.44
	(196.11)	(0.30)
Constant	3, 984.51***	5.88***
	(838.07)	(1.03)
Observations	224	237
Num users	31	31
Num tasks	14	14
sd(user)	1,489.25	0.82
sd(task)	1,104.7	1.14
R2m	0.004	0.008
R2c	0.642	0.289
Akaike Inf. Crit.	3,987.14	1,106.66
Bayesian Inf. Crit.	4,007.61	1,127.46

Note: *p < 0.1; **p < 0.05; ***p < 0.01.

of mixed-effects regression models and their specific designs, we note the standard threats to statistical conclusion validity affecting linear regression models in general. To mitigate these, we take standard precautions. First, we removed as outliers the top 1% most extreme values. Second, we checked for collinearity among the predictors we use the **variance inflation factor (VIF)** [22]; all were below 3, i.e., multicollinearity is not an issue [58]. Finally, we acknowledge that additional time may be spent as the users are asked to upload their edits, increasing the amount of time necessary in the plugin setting. However the time spent for uploading is minimal as the plugin automatically helps the user to remove the auto-generated comments with only a press of a keyboard shortcut.

Results. Table 3 summarizes our default specification mixed-effects regressions for both response variables; the models with our second specification (de-meaned task experience) are equivalent (see Appendix G). All models include controls for the amount of users' experience with the respective task categories as well as other random user and task effects. In all cases, the models fit the data reasonably well (ranging from $R^2_c = 29\%$ for task correctness scores, to $R^2_c = 64\%$ for task completion time), with most of the variance explained attributable to the two random effects (task and user)—there is significant user-to-user and task-to-task variability in all response variables.

Analyzing the models, we make the following observations: First, looking at the completion time model (1), there is no statistically significant difference between the two conditions. Stated differently, we do not find sufficient evidence to conclude that users in the plugin condition complete their tasks with different speed on average than users in the control group, contrary to our expectation.

Second, and this time in line with our expectation, there is no statistically significant difference between the two conditions in task correctness scores (model (2)). That is, the code written by users in the plugin condition appears statistically indistinguishably as correct from the code written by users in the control group.

We investigate more differences between the code written by study participants in each of the two conditions in more detail in the next section.

6 RQ_2: COMPARISON OF GENERATED VS. RETRIEVED CODE

In this section, we focus on *how* study participants are interacting with the code generation and retrieval systems. Specifically, we dive deeper into both the inputs to and the outputs of the plugin, i.e., we analyze the quality of the queries issued by study participants and of the code snippets produced in return, contrasting code generation to retrieval throughout. We analyze these data along three dimensions, detailed next.

6.1 For What Queries Do Users Tend to Favor Generation vs. Retrieval Answers

First, we investigate whether there are any discernible characteristics of the natural language queries (and therefore tasks) that associate with study participants tending to favor the code snippets returned by the code generation model over those returned by the code retrieval model.

Methodology. Using our instrumented environment, we collect all *successful* queries issued by the study participants, i.e., those for which a code snippet from among the listed candidates was selected, and we record which of the two sources (generation or retrieval) the snippet came from. See Table 10 in Appendix H for the complete set of queries from our 31 participants, organized per task. We then build a binary logistic regression model with snippet source as outcome variable and bag-of-words features of the natural language input queries as predictors.

If this model is able to predict the source of the code snippet better than by chance, then we can conclude that there is some correlation between the type of input query and the users' preference for generated versus retrieved code snippets. Moreover, the word feature weights in the logistic regression model could shed some light on what features are the most representative of queries that were effectively answered using generation or retrieval. For our analysis, we manually review the top 20 (approximately 7%) contributing query features for each value of the outcome variable ("generation" vs. "retrieval") and discuss patterns we observe qualitatively, after thematic analysis.

Specifically, for each query, we tokenize it, filter out English stop words, and compute a bag-of-words and bag-of-bigrams vector representation, with each element of the vector corresponding to the number of times a particular word or bigram (two-word sequence) occurred in the query. The number of distinct words in all queries is 302, and the number of distinct bigrams in all queries is 491, and thus the dimensionality of the query vector is 793.[29] We then estimate the model:

$$Pr(\text{chosen snippet is "generated"}) = \frac{\exp(\mathbf{X}\beta)}{1 + \exp(\mathbf{X}\beta)}, \tag{3}$$

where $\mathbf{X}$ here represents a k-dimensional bag-of-word vector representation of a given query, and β are the weights to be estimated. To this end, we randomly split all the collected query and candidate selection pairs into training (70% of the data) and held-out test (30%) sets. We then train the model using 5-fold cross-validation until it converges, and subsequently test it on the held-out set. We use 0.5 as a cutoff probability for our binary labels. In addition, we also build a trivial baseline model that always predicts "retrieval."

The baseline model is 55.6% accurate (among the successful queries in our sample there are slightly more code snippets retrieved rather than generated). Our main logistic regression model is 65.9% accurate, i.e., the model was able to learn some patterns of differences between those queries that result in code generation results being chosen over code retrieval ones and vice versa.

[29] We also experimented with other features, e.g., query length, query format compliance, and so on, but did not notice a significant difference in prediction accuracy.

Threats to Validity. One potentially confounding factor is that the plugin always displays code generation results first, before code retrieval. Ordering effects have been reported in other domains [102] and could also play a role here. Specifically, users who inspect query results linearly, top-down, would see the code generation results first and might select them more frequently than if the results were displayed in a different order. That is, we might infer that users prefer code generation to retrieval only because they see code generation results first, thus overestimating the users' preference for code generation versus retrieval.

Even though testing ordering effects experimentally was not practical with our study design, we could test a proxy with our log data—to what extent the code generation results overlap with the code retrieval ones. High overlap could indicate that code retrieval results might have been chosen instead of code generation ones, if presented earlier in the candidates list. Whenever study participants chose a snippet returned by the code generation model, we compared (as strings) the chosen snippet to all candidates returned by the code retrieval engine. Only 6 out of 173 such unique queries (~3.5%) also contained the exact chosen code generation snippet among the code retrieval results, therefore, we conclude that this scenario is unlikely.[30]

Another potentially confounding factor is that an icon indicative of generation or retrieval is displayed next to each result in the plugin UI. This means that users know which model produced which candidate snippet and might choose a snippet because of that reason rather than because of the snippet's inherent usefulness. More research is needed to test these effects. We hypothesize that biases may occur in both directions. On the one hand, holding other variables like ordering fixed, users might prefer code generation results because of novelty effects. On the other hand, users might prefer code retrieval results because of general skepticism towards automatically generated code, as has been reported, e.g., about automatically generated unit tests [33, 103].

Regarding the analysis, we use an interpretable classifier (logistic regression) and follow standard practice for training and testing (cross-validation, held-out test set, etc.), therefore, we do not expect extraordinary threats to validity related to this part of our methodology. However, we do note the typical threats to trustworthiness in qualitative research related to our thematic analysis of top ranking classifier features [88]. To mitigate these, we created a clear audit trail, describing and motivating methodological choices, and publishing the relevant data (queries, top ranking features after classification, etc.). Still, we note potential threats to transferability that may arise if different features or different classifiers are used for training, or a different number/fraction of top ranking features is analyzed qualitatively for themes.

Results. In Table 4, we show the top features that contributed to predicting each one of the two categories, and their corresponding weights. Inspecting the table, we make two observations:

First, we observe that for code generation, the highest ranked features (most predictive tokens in the input queries) refer mostly to basic Python functionality, e.g., "open, read csv, text file" (opening and reading a file), "sort, list, number, dictionary, keys" (related to basic data structures and operations in Python), "random number" (related to random number generation), "trim" (string operations), and so on. For example, some stereotypical queries containing these tokens that result in the code generation snippets being chosen are "open a csv file `data.csv` and read the data," "get date and time in gmt," "list all text files in the data directory," and so on.

In contrast, we observe that many queries that are more likely to succeed through code retrieval contain terms related to more complex functionality, some usually requiring a series of steps to fulfill. For example, "datetime" (regarding date and time operations), "cross validation, sklearn, column csv" (regarding machine learning and data analysis), "matplotlib" (data visualization), and

[30]Note that this only considers exact substring matches. There may be additional instances of functionally equivalent code that is nonetheless not an exact match.

Table 4. Most Important 20 Features and Their Weights from the Logistic Regression Modeling Whether Successful Plugin Queries Result in Generated or Retrieved Code Snippets

Generation				*Retrieval*			
Weight	Feature	Weight	Feature	Weight	Feature	Weight	Feature
0.828	open	0.352	current	0.471	letters	0.294	extract
0.742	time	0.345	delete row	0.442	copy	0.289	set
0.676	sort	0.345	random number	0.438	matplotlib	0.289	plt set
0.590	read csv	0.339	trim	0.437	datetime	0.282	read file
0.556	list	0.330	text file	0.410	python	0.282	cross-validation
0.507	number	0.326	keys	0.365	column csv	0.274	scikit
0.402	search	0.310	round	0.361	bar	0.274	dataframe csv
0.399	open file	0.293	numbers	0.344	copy files	0.274	sklearn
0.385	dictionary	0.291	row dataframe	0.334	delete column	0.272	digit
0.353	read	0.290	load csv	0.302	write file	0.270	folders

so on, are all among the top features for queries where users more often chose the code retrieval snippets.

In summary, it seems predictable (substantially more so than by random chance) whether natural language user queries to our NL2Code plugin are more likely to succeed through code generation vs. code retrieval on average, given the contents (words) of the queries.

6.2 How Well-specified Are the Queries

Search is a notoriously hard problem [47, 69], especially when users do not start knowing exactly what they are looking for, and therefore are not able to formulate clear, well-specified search queries. In this subsection, we investigate the quality of the input natural language queries, and attempt to delineate it from the quality of the underlying code generation and retrieval systems—either one or both may be responsible for failures to obtain desirable code snippets for a given task.

Anecdotally, we have observed that input queries to our NL2Code plugin are not always well-specified, even when the participants selected and inserted into their code one of the candidate snippets returned by the plugin for that query. A recurring issue seems to be that study participants sometimes input only a few keywords as their query (e.g., "move file"), perhaps as they are used to interacting with general purpose search engines like Google, instead of more detailed queries as expected by our plugin. For example, study participants sometimes omit (despite our detailed instructions) variable names part of the intent but defined elsewhere in the program (e.g., "save dataframe to csv" omits the DataFrame variable name). Similarly, they sometimes omit flags and arguments that need to be passed to a particular API method (e.g., "load json from a file" omits the actual JSON filename).

Methodology. The key idea behind our investigation here is to replace the underlying code generation and retrieval systems with an oracle assumed to be perfect—a human expert Python programmer—and study how well the oracle could have produced the corresponding code snippet given a natural language input query. If the oracle could successfully produce a code snippet implementing the intent, then we deem the query "good enough," or well-specified; otherwise, we deem the query under-specified. The fraction of "good enough" queries to all queries can be considered as an upper bound on the success rate of a perfect code generation model.

Concretely, we randomly sampled 50 queries out of all successful queries issued during the user study (see Table 11 in Appendix I for the sample) and had the first author of this article, a

proficient programmer with eight years of Python experience, attempt to generate code based on each of them. The oracle programmer considered two scenarios: (1) generating code given the input query as is, without additional context; (2) if the former attempt failed, then generating code given the input query together with the snapshot of the source file the study participant was working in at the time the query was issued, for additional context.

For each query, we record three binary variables: two indicating whether each of the oracle's attempts succeeded, without and with additional context, respectively,[31] and the third indicating whether the code snippet actually chosen by the study participant for that query came from the code generation model or the code retrieval one; see Table 11 in Appendix I.[32]

We then measure the correlation, across the 50 queries, between each of the two oracle success variables and the code snippet source variable, using the phi coefficient ϕ [23], a standard measure of association for two binary variables similar to the Pearson correlation coefficient in its interpretation. This way, we can assess how close the code generation model is from a human oracle (the *good enough as is* scenario) and whether contextual information from the source code the developer is currently working on might be worth incorporating into code generation models in the future (the *good enough with context* scenario); note that the code generation model we used in this study [117, 124] does not consider such contextual information.

Threats to Validity. We follow standard practice for the statistical analysis in this section, therefore, we do not anticipate notable threats to statistical conclusion validity. Due to the limitations of our telemetry system, we did not record unsuccessful queries (i.e., queries that the user entered but no candidate is selected). As a result, queries that favor neither generation nor retrieval cannot be compared. However, we acknowledge three other notable threats to validity. First, we used only one expert programmer as oracle, which may introduce a threat to construct validity given the level of subjectivity in determining which queries are "good enough." To mitigate this, we discussed among the research team, whenever applicable, queries for which the expert programmer was not highly confident in the determination. Second, our random sample of 50 queries manually reviewed by the expert programmer is only representative of the population of 397 queries with 95% confidence and 13% margin of error, which may introduce a threat to internal validity. However, the relatively small sample size was necessary for practical reasons, given the high level of manual effort involved in the review. Finally, we note a potential threat to construct validity around the binary variable capturing the source (generation or retrieval) of the candidate code snippets selected by the study participants. There is an implicit assumption here that study participants know what the right answer (code snippet) should be given a natural language query and are able to recognize it among the candidates provided by the NL2Code plugin; therefore, we assume that the snippet source variable captures actual quality differences between code snippets produced by the generation and retrieval models, respectively. However, this may not be the case. To test this, we reviewed all the candidate snippets returned by the plugin for the first 6 among the 50 queries analyzed. Across the $6 \cdot 2$ models (generation/retrieval) $\cdot\, 7$ candidates per model $= 84$ candidate snippets, we only discovered one case where the study participant could have arguably chosen a more relevant

[31] The former implies the latter but not vice versa.

[32] Note that on the surface, when looking at the data in Table 11, the values of the former two binary variables (the oracle's determination) may not always seem intuitive given the query. For example, the oracle determined the query "pandas to csv" to be *not good enough*, even with context, while the query "pandas output csv," seemingly equivalent, was found to be *good enough with context*. In both cases, the intent appears to be exporting a pandas dataframe (a popular data science Python library) as a csv file. However, in the first example the snapshot of the source file the study participant was working in at the time of the query did not yet include *any* such dataframe objects; the user appears to have issued the query ahead of setting up the rest of the context. A context-aware code generation model would also not be able to extract any additional information in this case, similarly to the human oracle.

Table 5. Contingency Tables for the Two Oracle Comparison Scenarios in Section 6.2

Snippet	*Query*			
Generation	Good enough as is		Good enough w/ context	
	False	True	False	True
False	23	8	15	16
True	7	12	1	18

See Table 10 in Appendix H for the actual queries.

snippet. Therefore, we expect the incidence of violations of this assumption to be rare enough to not materially affect our results.

Results. Table 5 shows contingency tables for each of the two oracle comparison scenarios. Note that the "good enough with context" category includes all queries that are "good enough as is," by construction. Inspecting the results in the table, we make the following observations:

First, the natural language queries analyzed are more often than not insufficiently well-specified for even the human expert to be able to write code implementing those intents; only 20 out of 50 queries (40%) are deemed "good enough as is" by the oracle. Representative examples of failures from Table 11 are the queries consisting of a few keywords (e.g., "csv writer," "defaultdict") rather than queries containing sufficient details about the user's intent (e.g., "remove first column from csv file"). Considering the source file the user was editing at query time helps, with 34 (68%) of the queries now being deemed "good enough with context" by the oracle.

Second, there is moderately high and statistically significant association between the success of the code generation model (i.e., the study participant choosing one of those candidate code snippets) and the quality of queries in both scenarios: $\phi = 0.37$ ($p = 0.008$) for already well-specified queries and $\phi = 0.45$ ($p = 0.001$) for queries that become informative enough given additional context. This suggests that input query quality can have a big impact on the performance of the code generation model, and that incorporating additional contextual information may help.

Analyzing the failure rate of the code generation model (generation = False), we observe that it is relatively high in general (31 out of 50 queries, or 62%). However, most of these cases are in response to under-specified queries (23 out of the 31 failures; 74%), for which even the human oracle failed to generate the corresponding code. Still, there are 8 (26%) failure cases where the human expert could directly implement the natural language intent without additional context: "date now," "for loop on range 100," "generate random letters," "get now one week from now," "get time and date," "open "data.csv" file," "how to remove an item from a list using the index," and "plt create 3 subplots." All but the last one seem to refer to basic Python functionality. These queries are targets where further improved code generation techniques could improve the utility of the plugin.

Interestingly, we also observe a non-trivial number of under-specified queries (7 out of 30; 23%) for which the code generation model succeeded despite the human oracle failing: "call `pick_with_replacement`," "copy a file to dist," "pandas round value," "pandas to csv," "rename column pandas," "plt ax legend," and "scatter."

6.3 How Much the Code Snippets Are Edited after Plugin Use

Choosing (and inserting into the IDE source file) one of the candidate code snippets returned by the NL2Code plugin indicates that the code snippet was generally useful. However, while useful, the code snippet may still be far from an ideal solution to the user's query. To get a sense of how

appropriate the accepted code snippets are given the user intent, we compare the distributions of snippet lengths before (i.e., as returned by the plugin) and after potential edits in the IDE.

Methodology. When inserting a code snippet a user selected from among the plugin-returned candidates, we also insert special code comments in the source file around the snippet to mark the start and end of the code fragment corresponding to that particular intent (as shown in Figure 3). Study participants are instructed to use a certain key combination when they are done editing that code fragment to remove the delimiters and submit the edited version of the code fragment back to our server. Our analysis in this section compares the length of code snippets and types of tokens present between these two versions.

Specifically, we first tokenize and tag each version of a code snippet using a Python tokenizer and then compare the pairs of distributions of lengths before and after edits for code snippets originating from each of the two underlying models, generation and retrieval, using the non-parametric Wilcoxon signed-rank test; in addition, as a measure of effect size, we compute the median difference between members of the two groups, i.e., the Hodges–Lehman estimator [46]. We also compute and report on the Levenshtein edit distance between the two versions, in terms of number of tokens. Figure 6 visualizes these different distributions.

Threats to Validity. We note two potential threats to construct and external validity related to the analysis in this section. First, we have no way of enforcing that study participants contain their code edits related to a particular intent to the section of the source file specially delimited by code comments for this purpose. One may include unrelated edits in the same code region or make related edits outside of the designated region. Therefore, our measurement of *snippet length post edits* may not accurately reflect the construct of snippet length as related to a particular intent. To mitigate this, we gave clear instructions to participants at the beginning of the study and manually reviewed a small sample of the edited versions of a snippet, not discovering any obvious noise. Second, not all study participants followed our instructions every time they used the plugin and submitted their final (edited or not) version of the snippet back to our server. Only 303 out of the 397 successful queries recorded (76.3%) had final code snippets uploaded back to our server. Since this was not a random sample, our findings on this sample may not generalize to the entire population of 397 successful queries. To assess the severity of this potential threat, we compared the distributions of plugin-returned code snippet lengths between all successful queries and just the 303 queries where study participants uploaded their edits onto our server; for both generated (Wilcoxon $p = 0.54$) and retrieved ($p = 0.93$) code snippets, we found the respective two distributions statistically indistinguishable, therefore, we expect this to not be a sizable threat to validity.

Results. Comparing the two distributions of token lengths for acceptable code snippets from the code generation model before and after edits, we do not find any statistically significant differences in their mean ranks ($p = 0.345$). The mean edit distance between the two versions of these snippets is 5.2 tokens (min 0, max 130, median 1).

In contrast, comparing the two distributions of token lengths for acceptable code snippets from the code retrieval engine before and after edits, we find a statistically significant difference in their mean ranks ($p = 1.195\text{e}{-7}$). The Hodges–Lehman median difference between the edited and unedited versions of these snippets is 18 tokens, with a 95% confidence interval from 11 to 23 tokens. The edit distance metric paints a similar picture—acceptable code snippets from the code retrieval engine, before and after edits, are at a mean edit distance of 13.2 tokens from each other (min 0, max 182, median 0).

We also note that code retrieval snippets tend to be longer than code generation ones both before ($p < 2.2\text{e}{-16}$; median difference 18 tokens, with a 95% confidence interval from 14 to Infinity) and after edits ($p = 2.657\text{e}{-14}$; median difference 10 tokens, with a 95% confidence interval from 7 to

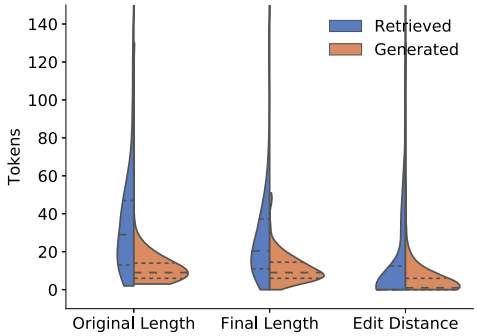


Fig. 6. Split violin plots comparing the length (in tokens) of the code snippets chosen by the study participants across all successful queries, before and after potential edits in the IDE. The horizontal dotted lines represent 25% and 75% quartiles, and the dashed lines represent medians.

Infinity). This may help explain why the retrieved snippets require more edits to correct the code to better suit the current programming code context, compared to the generated snippets.

Diving deeper into the edits to the plugin-supplied version of the different snippets, we compute the frequency distribution of tokens in both versions (plugin and final), normalized based on total token count in each corpus. Table 6 highlights the tokens with the greatest increases and decreases in relative frequency during editing. We observe that study participants seem to add common keywords such as "in, for, if, with," built-in names and functions such as "key, print," and common variable names such as "line, filename" to the generated/retrieved candidates. Stated differently, in these cases the code snippets seem to miss substantive parts and relevant functionality, which also may be partly due to the lack of specificity described in the previous section.

In contrast, study participants seem to delete number and string literals from the code snippets. This may be explained by the fact that the tool used retrieved code snippets as they appeared on Stack Oveflow, and thus many retrieved code snippets contain additional boilerplate code required for initialization or setup and hard-coded example inputs and outputs. We also observe some commonly used variable names like "df, plt" that get deleted, suggesting that variable replacement is one of the common operations when reusing the code snippets. An interesting observation here is that "In" and "Out" are getting deleted frequently. We find that it is mostly due to some of the code snippets retrieved from Stack Overflow being in the format of IPython REPL, which uses "In" and "Out" to separate the Python source code and execution outputs. When integrating these snippets, the users will have to remove this superfluous text. Figure 7 shows a representative example of such user edits after selecting a candidate snippet, which involves deleting IPython REPL contents, variable replacement and addition, as well as literal replacements.

Furthermore, following the previous observations on actual tokens, we are interested in how the frequency of different *types* of tokens changes before and after users edit the plugin-returned code snippets. We use the `tokenize`[33] Python 3 library to parse and tag the code snippets and compare the frequency changes by token type, similar to the previous analysis.[34] The results are shown in Table 7. We find that users add new `NAME` (identifiers, keywords) tokens the most, with the frequency of `STRING` (string literal) tokens slightly increased, and `COMMENT` (comment strings)

[33] https://docs.python.org/3/library/tokenize.html.

[34] Three of the retrieved snippets cannot be parsed and thus are omitted. See full explanation of different token types at https://www.asmeurer.com/brown-water-python/tokens.html. We also left out some uninteresting token types, such as `ENCODING`, `ENDMARKER`, `NL`.

Table 6. Most Frequently Added/Deleted Tokens after User Edits to Plugin-returned Code Snippets

Addition				*Deletion*			
ΔFreq.	Token	ΔFreq.	Token	ΔFreq.	Token	ΔFreq.	Token
0.0040	in	0.0016	w	−0.0071	2	−0.0016	In
0.0037	for	0.0015	with	−0.0071	1	−0.0016	11
0.0030	line	0.0015	``	−0.0043	a	−0.0015	y
0.0024	file	0.0015	days	−0.0038	0	−0.0014	Seattle
0.0023	key	0.0015	cur_v	−0.0034	3	−0.0014	12
0.0023	os.path.join	0.0015	company_info	−0.0025	plt	−0.0013	4
0.0021	dic	0.0015	n	−0.0023	50	−0.0013	iris
0.0021	filename	0.0014	output	−0.0021	id_generator	−0.0013	string.digits
0.0018	print	0.0014	codecs.open	−0.0018	Out	−0.0013	10
0.0017	if	0.0014	v	−0.0017	df	−0.0013	matplotlib.pyplot

Table 7. Frequency Changes of Different Token Types after User Edits to Plugin-returned Code Snippets

ΔFreq.	Type	ΔFreq.	Type	ΔFreq.	Type	ΔFreq.	Type
0.0138	NAME	0.0053	DEDENT	0.0004	COMMENT	−0.0095	OP
0.0053	INDENT	0.0022	STRING	−0.0049	NEWLINE	−0.0248	NUMBER

Sorted in descending order, positive number represents addition and negative number represents deletion.

Unedited

```
In [479]: df
Out[479]:
    ID  birthyear   weight
0   619040  1962    0.123123
1   600161  1963    0.981742
2   25602033    1963    1.312312
3   624870  1987    0.942120

In [480]: df["weight"].mean()
Out[480]: 0.83982437500000007
```

Edited

```
car_prices = car_prices["price"].mean()
```

Fig. 7. Representative example of user edits to a code snippet retrieved from Stack Overflow.

tokens staying roughly the same after the edits. NUMBER (numeric literal) tokens are deleted the most, in line with the observation above, again suggesting that many plugin-returned snippets are not tailored to specific identifiers and parameters that the user desires. Interestingly, we also see a slight decrease in frequency of NEWLINE tokens, representing a decrease in the number of logical lines of Python code after edits. This suggests that the plugin-returned code snippets are not concise enough in some cases.

7 RQ_3: USER PERCEPTIONS OF THE NL2CODE PLUGIN

Our last research question gauges how study participants perceived working with the NL2Code plugin, their pain points, and their suggestions for improvement.

Methodology. As part of our post-test survey, we asked the participants open-ended questions about what worked well when using the plugin and, separately, what they think should be improved. In addition, we asked participants to rate their overall experience using the plugin on a

Likert scale, ranging from 1 (very bad) to 5 (very good). We then qualitatively coded the answers to open-ended questions to identify themes in the responses for the 31 who completed all their assigned tasks.

Threats to Validity. We acknowledge usual threats to trustworthiness and transferability from qualitatively analyzing a relatively small set of open-ended survey data [88], as also discussed above. In particular, we note that only one researcher was involved in coding. To mitigate these threats, we release all verbatim survey responses as part of our replication package.

Results. Overall, study participants report having a neutral (15/31; 48.4%) or at least somewhat positive (15/31; 48.4%) experience using the NL2Code plugin, with only one participant rating their experience as somewhat negative.

Among the aspects the participants report as **positive**, we distill two main themes:

The plugin helps find code snippets the developer is aware of but cannot fully remember. (P1, P2, P8, P10, P11, P19, P20, P21, P22, P30, P31) These tend to be small commands or less familiar API calls and API usage patterns that users have seen before. Two participants summarize this well:

> "On a few occasions, the plugin very conveniently gave me the snippet of code I was looking for, [which] was "on the tip of my tongue." (P10)

> "Sometimes I just cannot remember the exact code, but I remember the shape. I could select the correct one easily." (P2)

Respondents expressed appreciation for both the generation and retrieval results, and there was little expression of preference for one method over the other, e.g.:

> "Even just having the snippets mined from Stack Overflow visible in the IDE was a good memory refresher / source of ideas." (P10)

> "It was somewhat convenient to not have to switch tabs to Google things, ..., based on my memory, that most of the suggestions I got were from the internet anyway." (P5)

> "It has all resources needed at one place." (P6)

Using an in-IDE plugin is less disruptive than using a web browser. (P1, P4, P5, P6, P7, P10, P18, P20, P24, P27) Many of our respondents who were positive about the plugin reiterate expected context-switching benefits of not leaving the IDE while programming, e.g.:

> "I like that the plugin stops me having to go and search online for solutions. [...] It can be very easy to get distracted when searching for solutions online." (P20)

> "Compared with manual search, this is faster and less disruptive." (P1)

Participants also describe many aspects of the plugin that **could be improved**.

The quality of code generation and retrieval results could be higher. (P3, P4, P5, P7, P9, P13, P14, P23, P27, P29, P31) Respondents mentioned that it was "rare" (P7) when they could directly use code from plugin, without modifications. In some cases, results from the plugin were "not related to the search" (P14), and users "didn't find what [they were] searching for" (P31). As one respondent humbly summarized it:

> "The model needs some improvements." (P4)

The insufficient quality of the plugin's results was especially felt as the tasks became more complex and involved APIs with complex usage patterns. One participant summarized this well:

> "For easy tasks, like walking through a directory in the filesystem, the plugin saves me time because what I did previously was to go to Stack Overflow and copy the code. But for

> difficult tasks like data processing or ML, the plugin is not helpful. Most snippets are not useful and I have to go to the website of sklearn to read the full doc to understand what I should do." (P3)

A particular related pain point is that the snippets from the code retrieval engine often contain spurious elements (as also noted above). In one participant's words:

> "When inserting the code into my program, I would like to **not** copy the input/output examples, and I can't imagine ever wanting those in the program itself." (P5)

Users could benefit from additional context. (P3, P5, P8, P18, P19, P20, P24, P26, P27) Some respondents mention that it would be useful to include additional (links to) explanations and documentation alongside the returned code snippets so the user could understand what the snippet is supposed to do, or even "which of the suggestions is the correct one when you are not familiar with a module" (P11). In two participants' words:

> "It would be nice if the examples from the internet could contain the relevant context of the discussion (e.g., things to consider when using this suggestion), as well as the input/output examples." (P5)

> "I hope the generated code snippet can have more comments or usage [examples]. Otherwise I still need to search the web to understand what it is." (P3)

A closely related theme is that *using the plugin assumes one has a "good background understanding of the underlying principles/modules/frameworks"* (P11), and they primarily need help with "look[ing] up little syntax bits that you have forgotten" (P11). (P1, P11, P16, P25) One participant was especially critical:

> "For more complex problems, I think the plugin does not help at all, because the programmer needs to know the theoretical background." (P16)

The plugin could benefit from additional context. (P4, P9, P10, P17, P30) Some participants suggest that the plugin could be "smarter" if it becomes more aware of the local context in the developer's IDE, e.g.:

> "Sometimes I want to generate an expression to be inserted somewhere, to be assigned to a variable, or to match the indentation level, without having to tell the plugin this explicitly. I didn't feel like the plugin was aware of context." (P10)

Participants also comment on how *the plugin's query syntax takes some getting used to* (P2, P12, P15), referring in particular to the way the code generation model expects queries to include variables, while the web search code retrieval engine allows users to only use keywords. For example:

> "[It became] useful to me towards the end when I got the hang of it and could formulate questions in the correct way (which I feel is somewhat of a skill in itself)." (P15)

> "It is not very natural for me to 'instantiate' my questions, I mostly like to search [using] keywords or just a description of what I want to achieve." (P2)

Querying the plugin could be interactive. (P11, P20, P30) Finally, some participants suggest to make querying interactive, dialogue-based, rather than unidirectional. This could with refining queries until they are sufficiently well-specified, or to decompose complex functionality into smaller steps, e.g.:

> "A chatbot [...] could identify the rough area in which the user needs assistance, [and] could help narrow it down further, helping to pinpoint an exact solution." (P20)

8 DISCUSSION AND IMPLICATIONS

Recent years have seen much progress from machine learning and software engineering researchers developing techniques to better assist programmers in their coding tasks, which exploit the advancements in (deep) learning technology and the availability of very large amounts of data from Big Code repositories such as GitHub and Stack Overflow. A particularly promising research direction in this space has been that addressing the decades-old problem of "natural language programming" [26], i.e., having people instruct machines in the same (natural) language they communicate in with each other, which can be useful in many scenarios, as discussed in the Introduction. However, while excited about this research direction and actively contributing to it ourselves, we are also questioning whether the most impact from such work can be had by focusing primarily on making technological advancements (e.g., as we write this, a one-trillion parameter language model has just been announced [28], as only the most current development in a very rapidly evolving field) without also carefully considering how such proposed solutions can fit within the software development workflow, through human-centered research.

In this spirit, we have presented the results of a controlled experiment with 31 participants with diverse background and programming expertise, observed while completing a range of Python programming tasks with and without the help of a NL2Code IDE plugin. The plugin allows users to enter descriptions of intent in natural language, and have corresponding code snippets, ideally implementing said intent, automatically returned. We designed the plugin with two research goals in mind. First, we sought to evaluate, to our knowledge for the first time using a human-centered approach, the performance of some NL2Code *generation* model with state-of-the-art performance on a benchmark dataset, but unknown performance "in the wild." Second, we sought to contrast the performance and user experience interacting with such a relatively sophisticated model to those of a relatively basic NL2Code *retrieval* engine, which "merely" retrieves existing code snippets from Stack Overflow given natural language search queries. This way, we could estimate not only how far we are from not having to write *any* code while programming, but also how far we have come on this problem given the many recent advancements in learning and availability of datasets.

Main Results. Overall, our results are mixed. First, after careful statistical analysis in $\mathbf{RQ_1}$, comparing tasks completed with and without using the NL2Code plugin (and either of its underlying code generation or retrieval systems), we found no statistically significant differences in task completion times or task correctness scores.

The results for **code metrics (SLOC and CC)** can be seen as mixed. On the one hand, the code containing automatically generated or retrieved fragments is not, on average, any more complex or any less maintainable than the code written manually, insofar as the CC and SLOC metrics can distinguish. On the other hand, one could have expected the opposite result, i.e., that since NL2Code tools are typically trained on idiomatic code, using them should lead to "better," more idiomatic code overall, which might suggest lower SLOC and CC values, on average.

Among the possible explanations for why we do not find supporting evidence for the "better code" hypothesis, two stand out: (i) the two metrics are only crude approximations of the complex, multifaceted concept of code quality; and (ii) even when writing code "manually," developers still consult the Web and Stack Overflow (i.e., the same resources that these NL2Code tools were trained on) and copy-paste code therein. To better understand the interaction between using the plugin and using a traditional Web browser, we used the event logs from our instrumented environment and compared the distributions of in-browser Web searches between tasks where the 31 study participants used the NL2Code plugin (median 3, mean 5, min 0, max 35 searches per user per task) and tasks where they did not (median 4, mean 7, min 0, max 48). A mixed-effects regression model similar to the ones in Section 5, controlling for individual self-reported experience and with

random effects for `user` and `task`, reveals a statistically significant effect of using the plugin on the number of in-browser Web searches: On average, using the plugin is associated with 2.8 *fewer* in-browser Web searches; however, this effect is smaller than the standard deviation of the random `user` intercept (~4 in-browser Web searches). We conclude that developers still search the Web when using the plugin, even if slightly less than when not using the plugin.

Using a similar argument, the result for **task correctness scores** can be seen as mixed. Code containing automatically generated or retrieved snippets is not, on average, any less appropriate for a given task as per our rubric than code written manually. However, using the NL2Code plugin does not seem to help our study participants significantly improve their scores either, despite there being room for improvement. Even though across our sample the median score per task was 7 out of 10 when using the plugin and 6 when not using the plugin, the multivariate regression analysis did not find the difference to be statistically significant.

The result for **task completion times** can be seen as negative and, thus, is perhaps the most surprising of our results: On average, study participants do not complete their tasks statistically significantly faster when using the NL2Code plugin compared to when they are not using it. There are several possible explanations for this negative result. First, we acknowledge fundamental limitations of our study design, which we hope future researchers can improve on. In particular, our tasks, despite their diversity and, we believe, representativeness of real-world Python use, may not lend themselves sufficiently well to NL2Code queries and, therefore, study participants may not have sufficient opportunities to use, and benefit from, the plugin. Moreover, our study population (31 participants) may not be large enough for us to detect effects with small sizes, should they exist.

However, even with these limitations, considering also our results for $\mathbf{RQ_2}$ and $\mathbf{RQ_3}$, we argue that another explanation is plausible: *Our NL2Code plugin and its main underlying code generation technology, despite state-of-the-art (BLEU-score) performance on a benchmark dataset, is not developed enough to be markedly useful in practice just yet.* Our telemetry data ($\mathbf{RQ_2}$) shows not only that study participants still carry out in-browser Web searches even though the NL2Code plugin was available, as discussed above, but also that the code snippets returned by the plugin, when used, undergo edits after insertion in the IDE, suggesting insufficient quality to begin with. Our qualitative survey data ($\mathbf{RQ_3}$) paints a similar picture of overall insufficient quality of the NL2Code results.

Implications. While our study suggests that state-of-the-art learning-based natural language to code generation technology is ways away from being useful in practice, our results should be interpreted more optimistically.

First, we argue that **the problem is worth working on**. In contemporary software development, which involves countless and constantly changing programming languages and APIs, natural language can be a useful medium to turn ideas into code, even for experienced programmers. A large fraction of our study participants commended NL2Code developer assistants for helping them remember the precise syntax or sequence of API calls and their arguments, required to implement some particular piece of functionality. When integrated into the development workflow, e.g., through an IDE plugin, such systems can help developers focus by reducing the need for context switching, further improving their productivity. Our quantitative task performance results for the current version of this NL2Code plugin, while negative, do not imply that future, better performing such systems will also not be markedly useful in practice; the qualitative data from our our study participants already suggests otherwise, as does quantitative data from prior research on the usefulness of in-IDE code search plugins [92].

Second, we argue that **this particular style of code *generation* is worth working on**. Our analysis of input queries and resulting code snippets for $\mathbf{RQ_2}$ shows that the code generation model produces fundamentally different results than the (simple) code retrieval engine we used

for comparison, and that study participants choose snippets returned by the code generation model almost as frequently as they do snippets from the code retrieval engine. In turn, this suggests that, at least within the scope of the current study, one type of model cannot be used as a substitute for the other. As discussed above, the code generation model does almost always produce different results than the code retrieval model. However, it was unclear from that analysis whether the generated code snippets reflect some fundamentally higher level of sophistication inherent to the code generation model, or whether the code retrieval engine we used for comparison is simply too naive.

To further test this, we performed an additional analysis. Specifically, we looked up the chosen code generation snippets in the manually labeled Stack Overflow dataset used for training the code generation model to assess whether the model is simply memorizing the training inputs. Only 13 out of the 173 unique queries (~7.5%) had as the chosen code fragment snippets found verbatim in the model's training dataset. Therefore, the evidence so far suggests that the code generation model does add some level of sophistication, and customization of results to the developers' intent (e.g., composing function calls), compared to what *any* code retrieval engine could.

Third, we provide the following **concrete future work recommendations** for researchers and toolsmiths in this area, informed by our results:

- *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.

9 RELATED WORK

Finally, we more extensively discuss how this work fits in the landscape of the many other related works in the area.

9.1 NL2Code Generation

While we took a particular approach to code generation, there are a wide variety of other options. Researchers have proposed that natural language dialogue could be a new form of human-computer interaction, since nearly the advent of modern computers [26, 35, 44, 76]. The bulk of prior work either targeted **domain-specific languages (DSLs)**, or focused on task-specific code generation for general-purpose languages, where more progress could be made given the relatively constrained vocabulary and output code space. Examples include generating formatted input file parsers [63]; structured, idiomatic sequences of API calls [96]; regular expressions [60, 74, 90]; string manipulation DSL programs [100]; card implementations for trading card games [68]; and solutions to the simplest of programming competition-style problems [10].

With the recent boom of neural networks and deep learning in natural language processing, generating arbitrary code in a general-purpose language [123, 124] are becoming more feasible. Some have been trained on both official API documentation and Stack Overflow questions and

answers [117]. There are also similar systems[35] able to generate class member functions given natural language descriptions of intent and the programmatic context provided by the rest of the class [49], and to generate the API call sequence in a Jupyter Notebook code cell given the natural language and code history up to that particular cell [1].

9.2 NL2Code Retrieval

Code retrieval has similarly seen a wide variety of approaches. The simplest way to perform retrieval is to start with existing information retrieval models designed for natural language search and adapt them specifically for the source code domain through query reformulation or other methods [39, 45, 52, 71, 113, 115]. Other research works utilize deep learning models [4, 37, 47, 48] to train a relevance model between natural language queries and corresponding code snippets. It is also possible to exploit code annotations to generate additional information to help improve code retrieval performance [120] or extracted abstract programming patterns and associated natural language keywords for more content-based code search [52]. Many of the models achieve good performance on human-annotated relevance benchmark datasets between natural language and code snippets. Practically, however, many developers simply rely on generic natural-language search engines like Google to find appropriate code snippets by first locating pages that contain code snippets through natural language queries [104] on programming QA websites like Stack Overflow.

9.3 Evaluation of NL2Code Methods

To evaluate whether NL2Code methods are succeeding, the most common way is to create a "reference" program that indeed implements the desired functionality, and measure the similarity of the generated program to this reference program. Because deciding whether two programs are equivalent is, in the general case, undecidable [101], alternative means are necessary. For code generation in limited domains, this is often done by creating a small number of input-output examples and making sure that the generated program returns the same values as the reference program over these tests [15, 59, 114, 118, 126–130]. However, when scaling to broader domains, creating a thorough and comprehensive suite of test cases over programs that have a wide variety of assumptions about the input and output data formats is not trivial.

As a result, much research work on code generation and retrieval take a different tack. Specifically, many code generation methods [1, 49, 117, 123] aim to directly compare generated code snippets against ground truth snippets, using token sequence comparison metrics borrowed from machine translation tasks, such as BLEU score [89]. However, many code snippets are equivalent in functionality but differ quite largely in terms of token sequences, or differ only slightly in token sequence but greatly in functionality, and thus BLEU is an imperfect metric of correctness of a source code snippet [110].

Code retrieval, however, is the task of retrieving relevant code given a natural language query, which is related to other information retrieval tasks. Since code retrieval is often used to search for vague concepts and ideas, human-annotated relevance annotations are needed for evaluation. The common methods used in research work [37, 47, 121] compare the retrieved code snippet candidates given a natural language query, with a human-annotated list of code snippet relevance, using common automatic information retrieval metrics such as NDCG, MRR, and so on [73]. The drawback of this evaluation method is that the cost of retrieval relevance annotation is high and

[35]This is, of course, among the many other use cases for neural network models of code and natural language such as code summarization [48, 121] or embedding models that represent programming languages together with natural languages [30]. Allamanis et al. [3] provide a comprehensive survey of the use cases of machine learning models in this area.

often requires experts in the specific area. Also, since the candidate lists are usually long, only a few unique natural language queries could be annotated. For example, one of the most recent large-scale code search challenge CodeSearchNet [47] contains only 99 unique natural language queries, along with their corresponding code snippet relevance expert annotations, leading to smaller coverage of real-world development scenarios in evaluation.

Regardless of the automatic metrics above, in the end our final goal is to help developers in their task of writing code. This article fills the gap of the fundamental question of whether these methods will be useful within the developer workflow.

9.4 In-IDE Plugins

Similarly, many research works on deploying plugins inside IDEs to help developers have been performed. Both Ponzanelli et al. [91] and Ponzanelli et al. [92] focus on reducing context switching in IDE by incorporating Stack Overflow by using the context in the IDE to automatically retrieve pertinent discussions from Stack Overflow. Subramanian et al. [109] propose a plugin to enhance traditional API documentation with up-to-date source code examples. Rahman and Roy [97] and Liu et al. [70] design the plugin to help developers find solutions on the Internet to program exceptions and errors. Following the similar route, Brandt et al. [16] study opportunistic programming where programmers leverage online resources with a range of intentions, including the assistance that could be accessed from inside the IDE.

Besides plugin developed to reduce context-switching to other resources in developer workflows, Amann et al. [5] focus on collecting data of various developer activities from inside the IDE that fuel empirical research on the area [94].

This article proposes an in-IDE plugin that incorporates code generation in addition to code retrieval to test the user experience in the real development workflow. In the meantime it also collects fine-grained user activities interacting with the plugin as well as editing the code snippet candidates to provide public data for future work.

9.5 End-user Development

The direction of exploring using natural language intents to generate code snippets is closely related to end-user development [67], which allows end-users (people who are not professional software developers) to program computers. The work of Cypher et al. [24] is among the first that enables end-user to program by demonstration.

Traditionally, programming has been performed by software developers who write code directly in programming languages for the majority of functionality they wish to implement. However, acquiring the requisite knowledge to perform this task requires time-consuming training and practice, and even for skilled programmers, writing programs requires a great amount of time and effort. To this end, there have been many recent developments on no-code or low-code software development platforms that allow both programmers and non-programmers to develop in modalities of interaction other than code [105]. Some examples include visual programming languages such as Scratch [72], which offers a building-block style graphical user interface to implement logic. In specific domains such as user interface design and prototyping, recent advances in deep learning models also enable developers to sketch the user interface visually and then automatically generates user interface code with the sketch [14] or using existing screenshots [87].

Besides visual no-code or low-code programming interfaces, there has also been much progress on program synthesis [12, 29, 31, 108], which uses input-output examples, logic sketches, and so on, to automatically generate functions, with some recent advances that use machine learning models [10, 21, 27, 106]. Some works also generate programs from easier-to-write pseudo-code [59, 129].

There are other works in the area. Barman et al. [11], Chasins et al. [19, 20] make web automation accessible to non-coders through programming by demonstration, while [64–66] automate mobile applications with multimodal inputs including demonstration and natural language intents. Head et al. [43] combine teacher expertise with data-driven program synthesis techniques to learn bug-fixing code transformations in classroom scenarios. Head et al. [42] help users extract executable, simplified code from existing code. Ko and Myers [55, 56] provide a debugging interface for asking questions about program behavior. Myers and Stylos [82] discuss API designers should consider usability as a step towards enabling end-user programming. Kery et al. [53], Kery and Myers [54] enable data scientists to explore data easily with exploratory programming. Our article's plugin of using both state-of-the-art code generation and code retrieval to provide more natural programming experience to developers, with the potential future of enabling end-user programming, is related to Myers et al. [81], which envisions natural language programming.

9.6 Code Completion

Many developers use **Integrated Development Environments (IDEs)** as a convenient solution to help with many aspects during development. Most importantly, many developers actively rely on intelligent code-completion aid like IntelliSense[36] for Visual Studio [6, 94] to help developers learn more about the code, keep track of the parameters, and add calls to properties and methods with only a few keystrokes. Many of intelligent code-completion tools also consider the current code context where the developer is editing. With the recent advances in machine learning and deep learning, example tools such as IntelliCode[37] for Visual Studio, Codota,[38] and TabNine[39] present AI-assisted code-suggestion and code-completion based on the current source code context, learned from abundant amounts of projects over the Internet. The scope of our article is to investigate generating or retrieving code using natural language queries, rather than based on the context of the current source code.

10 CONCLUSION

In this article, we performed an extensive user study of in-IDE code generation and retrieval, developing an experimental harness and framework for analysis. This demonstrated challenges and limitations in the current state of both code generation and code retrieval; results were mixed with regards to the impact on the developer workflow, including time efficiency, code correctness, and code quality. However, there was also promise: Developers subjectively enjoyed the experience of using in-IDE developer management tools and provided several concrete areas for improvement. We believe that these results will spur future, targeted development in productive directions for code generation and retrieval models.

APPENDICES

A USER STUDY ENVIRONMENT DESIGN

To control the user study's development environment for different users as much as possible, and to enable data collection and activity recording outside the IDE (e.g., web browsing activity during the development), we design a complete virtual machine-based environment for users to access remotely and perform the user study on. We build the virtual machine based on a lot of open

[36] https://docs.microsoft.com/en-us/visualstudio/ide/using-intellisense.
[37] https://visualstudio.microsoft.com/services/intellicode.
[38] https://www.codota.com/.
[39] https://www.tabnine.com/.

source software, including Ubuntu 18.04 operating system[40] with XFCE 4.1 desktop environment.[41] The virtual machine software is VirtualBox 6.1.10,[42] and we use Vagrant software[43] for automatic virtual machine provisioning.

Inside the Linux virtual machine, we install and configure a set of programs for data collection and workflow control during the user study:

(1) **Python environment.** Python 3.6[44] is installed inside the VM, alongside with pip package manager and several commonly used Python packages for the user study tasks. The user is free to install any additional packages they need during the development.
(2) **IDE with plugin.** PyCharm Community Edition 2020.1, with the plugin described in Section 3 is installed. This provides consistent Python development environment for the user study and the testing of the code generation and retrieval. The plugin also handles various data collection processes inside the IDE.
(3) **Man-in-the-middle proxy.** We install `mitmproxy`[45] in the VM, along with our customized script sending logs back to our server. This infrastructure enables interception and data collection of both HTTP and secured HTTPS requests. With this, we can collect users' complete web browsing activities during the user study.
(4) **Web browser.** We install Firefox browser,[46] configured to use the proxy mentioned above so all users' browsing activities could be logged for analysis.
(5) **Keylogger.** We develop a program that runs in the background during the user study and logs all the user's keystrokes along with the timestamps to our server. With the keylogger, we can collect data outside the IDE about the users' activities. This data is useful for mining and analyzing developer activity patterns in terms of keyboard operations, for example, copy and pasting shortcuts.
(6) **User study control scripts.** We provide users a handful of scripts for easy and fully automatic retrieval, start and submission of the tasks. The scripts allow user to check their completion status of the whole study, as well as to pause and resume during a task for a break. All the user's task start, pause, resume, and submission events are logged so the completion time of each task for the user could be calculated.

B PRE-TEST SURVEY DETAILS

For each of the prospective participants, we asked them about two parts of the information in a pre-study survey, apart from personal information for contact purposes. The first is regarding programming experience, used to determine if the participants have enough expertise in Python as well as the categories of tasks that we designed. The questions are:

(1) Which of the following best describes your current career status: Student (computer science), Student (other field), Software Engineer, Data Scientist, Researcher, Other.
(2) How do you estimate your programming experience? (1: very inexperienced to 5: very experienced)
(3) How experienced are you with Python? (1: very inexperienced to 5: very experienced)

[40] https://releases.ubuntu.com/18.04/.
[41] https://www.xfce.org/.
[42] https://www.virtualbox.org/wiki/Downloads.
[43] https://www.vagrantup.com/.
[44] https://www.python.org/.
[45] https://mitmproxy.org/.
[46] https://www.mozilla.org/en-US/firefox/.

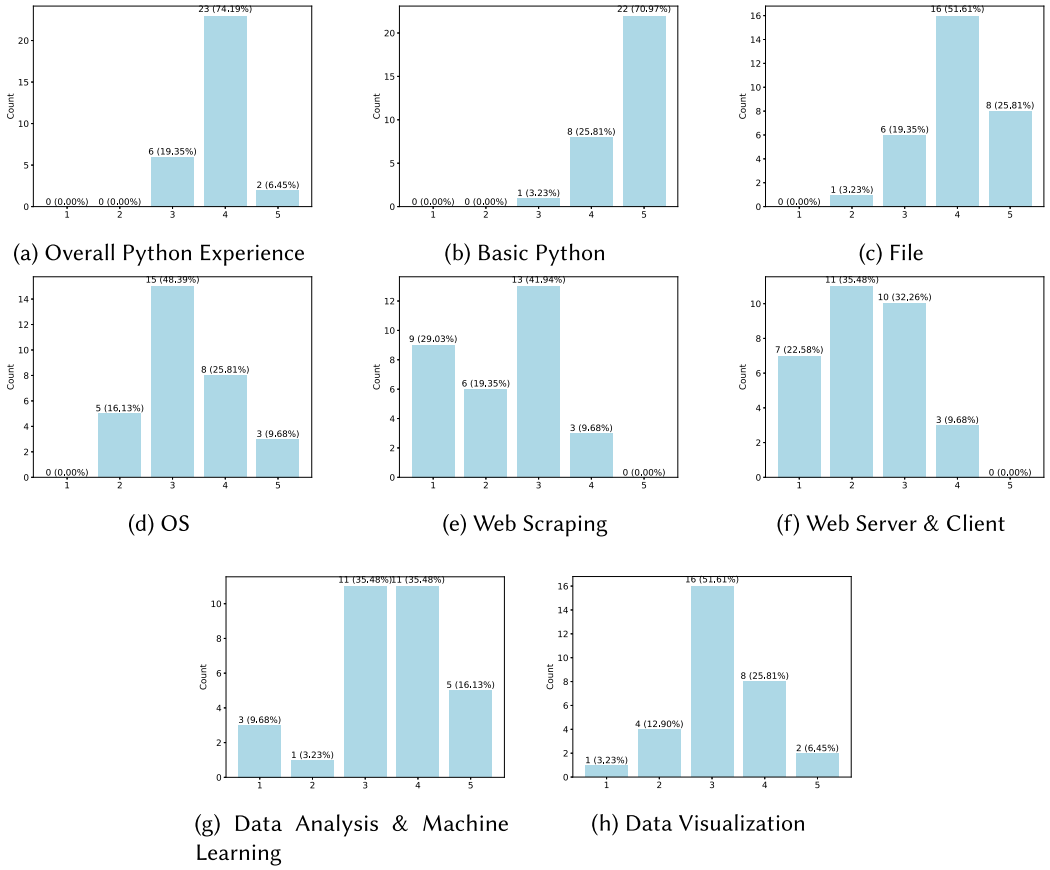
(4) How experienced are you with each of the following tasks in Python? (1: very inexperienced to 5: very experienced) Basic Python, File, OS, Web Scraping, Web Server & Client, Data Analysis & Machine Learning, Data Visualization.

The second part of the information is about their development preferences, used to ask for their preferences with IDE and assistive tools. The questions are:

(1) What editor/IDE do you use for Python projects? Vim, Emacs, VSCode, PyCharm, Jupyter Notebook, Sublime Text, other.
(2) Do you use any assistive tools or plugins to improve your coding efficiency? Some examples are code linting, type checking, snippet search tools, etc. If yes, what are they?

C PARTICIPANTS PROGRAMMING EXPERIENCE

The detailed participants' programming experience responded in the survey is shown in Figure 8.



(a) Overall Python Experience (b) Basic Python (c) File

(d) OS (e) Web Scraping (f) Web Server & Client

(g) Data Analysis & Machine Learning (h) Data Visualization

Fig. 8. The experience and expertise for overall Python programming and 7 specific areas that we design different tasks for, from all the participants that completed the survey. 1 represents very inexperienced and 5 represents very experienced.

D POST-STUDY SURVEY DETAILS

After each task, we ask the following questions to all users (disregarding using the plugin or not) about the task design, self-assessment, as well as the help needed during the process:

(1) How difficult did you feel about the task? (1: very easy to 5: very hard)
(2) How would you evaluate your performance on the task? (1: very bad to 5: very good)
(3) How often did you need to look for help during the task, including web search, looking up API references, etc.? (1: not at all to 5: very often)

For users that completed the current task with plugin enabled, the following additional questions about the plugin user experience are asked:

(1) How do you think the plugin impacted your efficiency timewise, if at all? (1: hindered significantly, to 3: neither hindered nor helped, to 5: helped significantly)
(2) How do you think the plugin impacted your quality of life, with respect to ease of coding, concentration, etc., if at all? (1: hindered significantly, to 3: neither hindered nor helped, to 5: helped significantly)

After all assigned tasks are completed for the user, we ask them to complete a form about the overall experience with the user study and the evaluation of the plugin, as well as soliciting comments and suggestions.

(1) What did you think of the tasks assigned to you in general?
(2) Overall, how was your experience using this plugin? (1: very bad to 5: very good)
(3) What do you think worked well, compared with your previous ways to solve problems during programming?
(4) What do you think should be improved, compared with your previous ways to solve problems during programming?
(5) Do you have any other suggestions/comments for the plugin?

E PLUGIN EFFECT ON CODE COMPLEXITY METRICS

We also analyze the plugin's effect on code complexity metrics, following the same methods used in Section 5. We measure two standard proxies for code complexity of the Python programs produced by our study participants in each of their assigned tasks, i.e., the number of **source lines of code (SLOC)** and McCabe's **cyclomatic complexity (CC)**, a measure of the number of linearly independent paths through a program's source code [75]; in real programs, CC depends a lot on the "if"-statements, as well as conditional loops and whether these are nested. The two measures tend to be correlated, but not strongly enough to conclude that CC is redundant with SLOC [61]. We use the open-source library Radon[47] to calculate CC.

One could expect that code produced by our NL2Code plugin may be more idiomatic (possibly shorter and less complex) than code written by the participants themselves.

Figure 9 shows the distributions of CC values across tasks and conditions. Figure 10 shows the distributions of SLOC values across tasks and conditions.

Table 8 summarizes our default specification mixed-effects regressions with CC and SLOC variables included; the models with our second specification (de-meaned task experience) are shown in Appendix G. The models fit the data reasonably well ($R^2_c = 50\%$ for SLOC, $R^2_c = 27\%$ for CC).

Analyzing the models, we make the following observations: There is no statistically significant difference between the two conditions in cyclomatic complexity values (model (4)). That is, the

[47] https://github.com/rubik/radon.

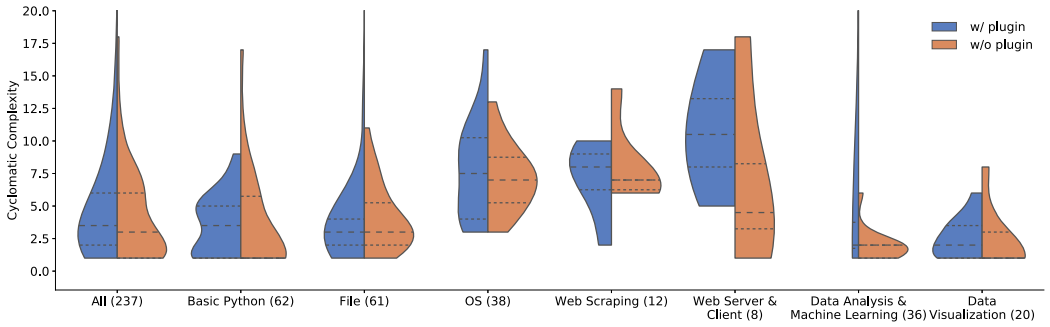


Fig. 9. Distributions of cyclomatic complexity values across tasks and conditions. The horizontal dotted lines represent 25% and 75% quartiles, and the dashed lines represent medians.

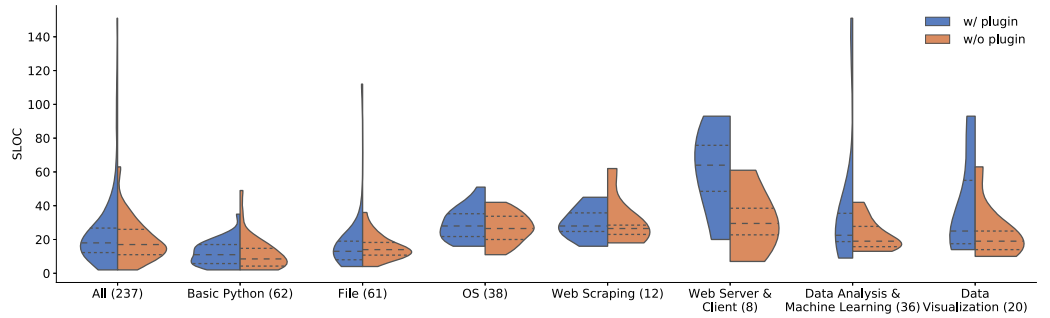


Fig. 10. Distributions of SLOC values across tasks and conditions. The horizontal dotted lines represent 25% and 75% quartiles, and the dashed lines represent medians.

code written by users in the plugin condition appears statistically indistinguishably as correct and as complex from the code written by users in the control group.

We note a small effect of using the plugin on code length (model (3)). On average, the code written by users in the plugin condition is ~4 source lines of code longer than the code written by users without using the plugin. However, this effect is quite small, smaller than the standard deviation of the random `user` intercept (~6 source lines of code).

F NL2CODE PLUGIN QUERY SYNTAX

For the best results from the code generation model, we also instruct the users to write queries as expected by the model with the following rules:

- Quote variable names in the query with grave accent marks: ... `variable_name` ...
- Quote string literals with regular quotation marks: ... “Hello World!” ...
- Example query 1: open a file “yourfile.txt” in write mode.
- Example query 2: lowercase a string `text` and remove non-alphanumeric characters aside from space.

Table 8. LMER Task Performance Models (Default Specification, w/code Complexity Metrics)

	Dependent variable			
	Completion time	Correctness score	SLOC	CC
	(1)	(2)	(3)	(4)
Experience	−195.62	0.07	−0.62	−0.21
	(183.11)	(0.24)	(1.61)	(0.46)
Uses plugin	15.76	0.44	4.16**	0.73
	(196.11)	(0.30)	(1.91)	(0.58)
Constant	3,984.51***	5.88***	27.15***	5.64***
	(838.07)	(1.03)	(7.40)	(1.95)
Observations	224	237	237	237
Num users	31	31	31	31
Num tasks	14	14	14	14
sd(user)	1,489.25	0.82	6.16	1.18
sd(task)	1,104.7	1.14	12.65	2.33
R2m	0.004	0.008	0.011	0.006
R2c	0.642	0.289	0.502	0.27
Akaike Inf. Crit.	3,987.14	1,106.66	2,002.42	1,417.27
Bayesian Inf. Crit.	4,007.61	1,127.46	2,023.23	1,438.08

Note: *p < 0.1; **p < 0.05; ***p < 0.01.

G TASK PERFORMANCE MODELS (DE-MEANED SPECIFICATION)

Table 9 summarizes our alternative specification (de-meaned task experience) mixed-effects regressions for two response variables in the main article, plug two response variables (CC and SLOC) introduced in Appendix E.

Table 9. LMER Task Performance Models (De-meaned Experience, w/code Complexity Metrics)

	Dependent variable			
	Completion time	Correctness score	SLOC	CC
	(1)	(2)	(3)	(4)
Experience BTW	−478.55	−0.04	−1.47	0.04
	(566.62)	(0.43)	(2.98)	(0.74)
Experience WI	−166.14	0.12	−0.30	−0.35
	(191.33)	(0.29)	(1.87)	(0.56)
Uses plugin	14.47	0.44	4.15**	0.74
	(196.07)	(0.30)	(1.90)	(0.58)
Constant	5,142.42**	6.32***	30.59**	4.62
	(2,348.61)	(1.77)	(12.60)	(3.07)
Observations	224	237	237	237
Num users	31	31	31	31
Num tasks	14	14	14	14
sd(user)	1,482.32	0.81	6.15	1.17
sd(task)	1,107.9	1.13	12.69	2.32
R2m	0.012	0.008	0.012	0.007
R2c	0.643	0.287	0.504	0.269
Akaike Inf. Crit.	3,988.86	1,108.56	2,004.30	1,419.09
Bayesian Inf. Crit.	4,012.74	1,132.84	2,028.58	1,443.36

Note: *p < 0.1; **p < 0.05; ***p < 0.01.

H USER QUERIES

Table 10. Unique Successful User Queries to the NL2Code Plugin, Per Task, for the 31 Study Participants

Task	Queries	
T1-1	**call `pick_with_replacement`**	**how to generate random letter**
	create a dictionary with keys `random_letters` and values `random_numbers`	import library random
	create dictionary	**list to dict**
	create empty dictionary	loop on numbers from 0 to 100
	create list "a_list"	**loop over a range of `count`**
	defaultdict	merge 2 dictionaries
	dictionary of characters and int	**pair characters in `characters` and numbers in `numbers`**
	for loop on range 100	**print `dic` keys on each line**
	generat integers 1–20	**print `dic` keys sorted**
	generate 100 integers (1–20 inclusive).	**print `dic` sorted by keys**
	generate 100 random lower-cased leters	print a to z
	generate 100 random lowercase letters	**print list**
	generate 100 random numbers	print list as string
	generate 100 random numbers from 1 to 20	print list elements
	generate a rondom lower case character	print without newline
	generate char lower case	random
	generate dict	random character between a and z
	generate list of random charachters	random characters
	generate lowercase char	**random integer between 1 and 20**
	generate random	**random number**
	generate random between 0 and 20	**random sample with replacement**
	generate random charachter	randomly generate 100 letters
	generate random int	**randomly pick an item from `seq`**
	generate random letters	**rearrange dictionary keys into alphabetic order**
	generate random lower case letters	**sort a list**
	generate random nu,ber	**sort a list into ascending order**
	generate random number	**sort a list x into ascending order**
	generate random numbers	**sort dict by key**
	generate random numbers between 1-20 inclusive	**sort key of dict**
	get a random letter	sort list
	given list `letters` and `integers`, create a dicitonary such that the values in `letters` are keys and values in `integers` are values	**sort list 'values' into ascending order**
	how to append value in dict	squence of integers from 1 to 20 inclusive
	how to check if a key is in a dictionay	zip 2 lists
	how to generate random int in range between 1 and 20	**zip `hundred_characters` with `hundred_numbers`**
T1-2	add a week to a datetime	get gmt timezone
	add days to time	get now one week from now
	assign current date and time to `now`	**get the current date in utc**
	change date format	**get the current time in utc**
	change datetime format of `week_date` to mm-dd-yyyy hh:mm	**get the date and time a week from now in gmt**
	convert `week_date` to GMT timezone and assign to `GMT_week_date`	get time and date
	convert date timezone	**get time and date in gmt in `date`**
	date from 7 days	**get time and date one week from now**
	date gmt	**get time now**
	date now	gmt
	datetime	gmt time 24
	display `week_date` in format mm-dd-yyyy hh:mm	import datetime
	format datetime	**import time**
	format datetime 24 hour	mm-dd-yyyy
	format time	**print current date time**
	get current datetime	**print date and time in GMT in 24hr format**
	get date 7 days from today	**print datetime in mm-dd-yyyy hh:mm format**
	get date and time in gmt	**time add**
	get date and time one week from now	time and date
	get date time one week from now	**time and date in certain**
	get datetime	**timedelta**

(Continued)

Table 10. (continued)

Task	Queries	
T2-1	**copy column from "data.csv" file to another "output.csv"**	**new line**
	copy column from "data.csv" to "output.csv"	**number of columns of csv**
	create 'output.csv' csv file	open "data.csv" file
	csv write	**open a csv file `data.csv` and read the data**
	csv writer	**open csv**
	cvs	**open csv file `data.csv`**
	cvs files	**open csv file with read and write**
	delete a column in csv	**open file**
	delete column from csv	**pandas read csv**
	delete column from csv file	**pandas read csv named "data.csv"**
	delete first and last column in csv file	print csv without row numbers
	delete first and last column of `df`	python make dir
	delete first and last row from the dataframe `df`	**read "data.csv" file**
	delete first row from dataframe `df`	**read csv file "data.csv"**
	delete row in csv	**read csv file using pandas**
	delete the first column in csv file `df`	read csv pure python
	file to csv	read cvs
	get current path	remove columns from csv file and save it to another csv file
	get specific columns by index in pandas data frame	**remove first column from csv file**
	headers in a dataframe	save `df` to a file `output.csv` in a new directory `example_output`
	how to delete a column in a dataframe python	**save dataframe to csv**
	how to delete columns in dataframe	save pandas dataframe to a file
	how to save a dataframe in csv file	**save this dataframe to a csv**
	if dir exist	**write `output` to csv file**
	if directory "output" exists	**write csv `output_f` to file "output/output.csv"**
	make directory	**write output to csv file "output.csv"**
	make directory "output" if it doesn't exist	**write to csv file**
T2-2	change directory	**list files in folder**
	change directory to "data"	**list of filenames from a folder**
	check file encoding	**move file to other directory**
	check if directory exists	**normalize newlines to \n**
	convert binary decoded string to ascii	**open file**
	convert file encoding	**open text file**
	convert file to utf	read a file and iterate over its contents
	convert latin-1 to utf-8	**read all files under a folder**
	convert str to utf-8	**read file**
	convert text file encoding	read ISO-8859-15
	convert text files from encoding ISO-8859-15 to encoding UTF-8.	readline encoding
	copy a file	redirect
	copy file	**remove header**
	copy file `ddd.png`	remove heading white space
	copy file to other folder	text normalize newlines to \n
	covert file to utf	traverse a directory
	find character	**travverse list of files**
	get all files in directory	**trim heading whitespace**
	get the file extension	**trim the heading and trailing whitespaces and blank lines for all text files**
	iterating files in a folder	unkown encoding
	list all text files in the data directory	write to file
	list files in directory	
T3-1	**check if `file` is a directory**	match regex year month day
	check if string has specific pattern	**move file**
	copy a file to dist	move files from directory to directory
	copy all files and directories from one folder to another	recursive copy files and folders
	copy directory to another directory	**recursively iterate over all files in a directory**
	copy directory to directory	**regex dd-mm-yy**
	copy directory tree from source to destination	regex digit python
	copy file from `src_path` to `dest_path`	regex for date
	copy files	**regex replace capture group**
	copy files and directories under `data` directory	regexp date
	copy files creating directory	rename file

(Continued)

Table 10. (continued)

Task	Queries	
	copy files from folder	**rename file with regex**
	create file	**rename files**
	create folder	replace pattern in string
	datetime to string	**search all matches in a string**
	extract year month day from string regex	**search for pattern "%d%d-%d%d" in `file`**
	get all files and folders	walk all files in a directory
	get the files that inside the folders	walk all nested files in the directory "data"
	list all filepaths in a directory	**walke all files in a directory**
	make a folder recersively	write to file
T3-2	**add entry to json file**	**load json file**
	check if file `output_file` exists	load json from a file
	check if file ends with .json	**read a json file named `f`**
	convert dict to string	sorting a dictionary by key
	convert list to dictionary	write into txt file
	import json parsing library	write json in `ret` to file `outfile`
T4-1	find all bold text from html `soup`	parse all hyperlinks from `r` using bs4
	find all hrefs from `soup`	visit `url` and extract hrefs using bs4
	find all red colored text from html `soup`	visit the given url `url` and extract all hrefs from there
	go to a url	**visit the url `url`**
	how to get page urls beautifulsoup	
T4-2	create directory	**regex []**
	download an image request	save dict to csv
	extract imafe from html	**save table beautifulsoup**
	http reques get html	
T5-1	**add json file to a list**	check email correctness
T5-2	argparse subprogram	**print format**
	exit program	request with params
	gET request to "https://jsonplaceholder.typicode.com/posts" with argument userId	
T6-1	a list of dictionary to pandas dataframe	pandas change dataframe column name
	add a new column to a dataframe row	pandas create buckets by column value
	average by group pandas	pandas dropnan
	cast a float to two decimals	pandas get average of column
	cast a list to a dataframe	pandas group by
	column to integer pandas	**pandas join dataframes**
	create a dataframe from a list	pandas join series into dataframes
	csv	pandas output csv
	csv write	pandas print with two decimals
	delete coloumn pd	**pandas read from csv**
	df set column to 7 decimals	**pandas round value**
	filter df with two conditions	pandas save csv two decimal
	filter values in pandas df	**pandas to csv**
	find unique data from csv	pandas to csv decimal
	findall	pandas write df to csv
	floating data in csv group in digit	pandas write to csv file
	format output to 2 decimal	pandas write to file decimal
	get average of row values in pandas dataframe	**read csv**
	get average value from group of data in csv	**read csv file**
	get the head of dataframe `df`	remove repeated column in csv file
	group by range pandas	**rename column pandas**
	group of data from csv	rename pandas df columns
	how to combine 2 lists into a dictionary	**round a variable to 2dp**
	how to remove an item from a list using the index	save `compan_df` dataframe to a file
	import pandas	save `compand_df` dataframe to a file
	list to an entry in pandas dataframe	**sort dataframe `jdf` by `scores`**
	load csv file with pandas	**sort dataframe `jdf` by the values of column 'scores'**
	loop files recursive	**sort pandas dataframe**
	newline space	standard deviation from group of data in csv
	pandas add new column based on row values	two deciaml place
	pandas calculate mean	**write `final_data` to csv file "price.csv"**
T6-2	cross validation in scikit learn	multinomial logistic regression model
	cross validation mean accuracy	**numpy load from csv**
	disable warnings	run 5-fold accuracy

(Continued)

Table 10. (continued)

Task	Queries	
	how to determine cross validation mean in scikit learn	set numpy random seed to 0
	how to split dataset in scikit learn	sklearn 5 fold cross validation
	how to split dataset in scikit learn	sklearn 5-fold cross validation
	linear regressor 5 folder cross validation	sklearn cross validation x, y for 5 folds
	load wine dataset	sklearn ignore warnings
T7-1	how to choose plot size in inches	plt set x axis tick range
	how to choose plot title in matplotlib	plt set xtick font size
	how to create ascatter plot using matplotlib	**reformat date**
	how to draw scatter plot for data in csv file	**save plot as image**
	plt create figure with size	**save plt figure**
	plt date as x axis	**scatter**
	plt set x axis label	scatter plot purple
T7-2	bar graph side by side	**plot bar**
	bar plot with multiple bars per label	plot size
	get height of bars in subplot bar gaphs	plot title
	get labels above bars in subplots	**plt ax legend**
	group pandas df by two columns	plt ax xlabel
	horizontal subplot	plt create 3 subplots
	import matplotlib	plt set title for subplot figure
	matplotlib grouped bar chart	plt set x tick labels
	matplotlib multiple histograms	plt show values on bar plot
	matplotlib theme	**pyplot subplots**
	pandas dataframe from csv	select row pandas
	pandas dataframe groupby column	

Queries for which the participant chose a snippet produced by the code generation model are shown in boldface, and in the remainder a retrieved snippet was used.

I RANDOMLY SAMPLED USER QUERIES FOR THE ORACLE ANALYSIS

Table 11. Sampled User Queries for the Oracle Analysis

Task	Queries	
T1-1	**call `pick_with_replacement` ○**	defaultdict
	generate lowercase char ●○	for loop on range 100 ●○
	generate random between 0 and 20 ●○	generate char lower case
	random sample with replacement ●○	generate random letters ●○
	sort key of dict ●○	random characters
T1-2	**change datetime format of `week_date` to mm-dd-yyyy hh:mm ●○**	format datetime
	convert `week_date` to GMT timezone and assign to `GMT_week_date` ●○	get gmt timezone ○
	print datetime in mm-dd-yyyy hh:mm format ●○	get now one week from now ●○
	date now ●○	get time and date ●○
T2-1	**remove first column from csv file ●○**	how to delete columns in dataframe ○
	csv writer	open "data.csv" file ●○
	how to delete a column in a dataframe python ○	
T2-2	traverse a directory ○	
T3-1	**copy a file to dist ○**	recursive copy files and folders ○
	match regex year month day	regexp date
T4-2	download an image request	save dict to csv
T5-2	**exit program ●○**	argparse subprogram
T6-1	**load csv file with pandas ●○**	how to remove an item from a list using the index ●○
	pandas round value ○	pandas create buckets by column value
	pandas to csv	pandas group by
	read csv file ●○	pandas output csv ○
	rename column pandas ○	pandas to csv decimal ○
	filter df with two conditions	pandas write df to csv
T6-2	load wine dataset	
T7-1	**plt create figure with size ●○**	**scatter ○**
T7-2	**plt ax legend ○**	plt create 3 subplots ●○
	bar plot with multiple bars per label ○	

Queries for which the user chose a snippet from the code generation model are shown in boldface. ● denotes queries "good enough" on their own; ○ denotes queries good enough given the rest of the source file as context; the former is a strict subset of the latter.

ACKNOWLEDGMENTS

We thank William Qian, who was involved in development of an early version of the plugin. We thank all participants who took part in the user study experiments for their effort on completing the tasks testing the intelligent programming interface. We would like to give special thanks to Ziyu Yao and NeuLab members Shuyan Zhou, Zecong Hu, among others, for the early testing of the plugin and the user study and their valuable feedback. We also thank anonymous reviewers for their comments on revising this article.

REFERENCES

[1] R. Agashe, Srini Iyer, and Luke Zettlemoyer. 2019. JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing (EMNLP/IJCNLP)*.

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *International Symposium on Foundations of Software Engineering (ESEC/FSE)*. 281–293.

[3] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.* 51, 4 (2018), 1–37.

[4] Miltiadis Allamanis, Daniel Tarlow, A. Gordon, and Y. Wei. 2015. Bimodal modelling of source code and natural language. In *32nd International Conference on Machine Learning (ICML)*.

[5] S. Amann, Sebastian Proksch, and S. Nadi. 2016. FeedBaG: An interaction tracker for Visual Studio. In *International Conference on Program Comprehension (ICPC)*. 1–3.

[6] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. 2016. A study of visual studio usage in practice. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 124–134.

[7] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. 1996. Query reformulation for dynamic information integration. *J. Intell. Inf. Syst.* 6, 2–3 (1996), 99–130.

[8] Philip Arthur, Graham Neubig, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Semantic parsing of ambiguous input through paraphrasing and verification. *Trans. Assoc. Comput. Ling.* 3 (2015), 571–584.

[9] Alberto Bacchelli, Luca Ponzanelli, and Michele Lanza. 2012. Harnessing stack overflow for the IDE. In *International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 26–30.

[10] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to write programs. In *5th International Conference on Learning Representations (ICLR)*.

[11] S. Barman, Sarah E. Chasins, Rastislav Bodík, and Sumit Gulwani. 2016. Ringer: Web automation by demonstration. In *ACM SIGPLAN International Conference on Object-oriented Programming, Systems, Languages, and Applications*.

[12] D. Basin, Y. Deville, P. Flener, A. Hamfelt, and Jørgen Fischer Nilsson. 2004. Synthesis of programs in computational logic. In *Program Development in Computational Logic*.

[13] Andrew Bell, Malcolm Fairbrother, and Kelvyn Jones. 2019. Fixed and random effects models: Making an informed choice. *Qual. Quant.* 53, 2 (2019), 1051–1074.

[14] Tony Beltramelli. 2018. pix2code: Generating code from a graphical user interface screenshot. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, 3:1–3:6. DOI: https://doi.org/10.1145/3220134.3220135

[15] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1533–1544.

[16] J. Brandt, P. Guo, J. Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *SIGCHI Conference on Human Factors in Computing Systems (CHI)*.

[17] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code snippet content assist via natural language tasks. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 628–632.

[18] Veronica Cateté and T. Barnes. 2017. Application of the delphi method in computer science principles rubric creation. In*ACM Conference on Innovation and Technology in Computer Science Education*.

[19] Sarah E. Chasins, S. Barman, Rastislav Bodík, and Sumit Gulwani. 2015. Browser record and replay as a building block for end-user web automation tools. In *24th International Conference on World Wide Web (WWW)*.

[20] Sarah E. Chasins, Maria Mueller, and Rastislav Bodík. 2018. Rousillon: Scraping distributed hierarchical web data. In *31st Annual ACM Symposium on User Interface Software and Technology (UIST)*.

[21] X. Chen, C. Liu, and D. Song. 2019. Execution-guided neural program synthesis. In *7th International Conference on Learning Representations (ICLR)*.

[22] J. Cohen. 2003. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Lawrence Erlbaum.

[23] Harald Cramér. 1999. *Mathematical Methods of Statistics*. Vol. 43. Princeton University Press.
[24] A. Cypher, Daniel C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. Myers, and Alan Turransky. 1993. Watch what I do: Programming by demonstration.
[25] M. Dawood, Khalid A. Buragga, Abdul Raouf Khan, and Noor Zaman. 2013. Rubric based assessment plan implementation for computer science program: A practical approach. In *IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE)*. 551–555.
[26] Edsger W. Dijkstra. 1979. On the foolishness of "natural language programming." In *Program Construction*. Springer, 51–53.
[27] K. Ellis, Maxwell Nye, Y. Pu, Felix Sosa, J. Tenenbaum, and Armando Solar-Lezama. 2019. Write, execute, assess: Program synthesis with a REPL. In *33rd Conference on Neural Information Processing Systems (NeurIPS)*.
[28] William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961* (2021).
[29] Y. Feng, R. Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[30] Zhangyin Feng, Daya Guo, Duyu Tang, N. Duan, X. Feng, Ming Gong, Linjun Shou, B. Qin, Ting Liu, Daxin Jiang, and M. Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
[31] John K. Feser, S. Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *36th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[32] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. CACHECA: A cache language model based code suggestion tool. In *International Conference on Software Engineering (ICSE)*. IEEE, 705–708.
[33] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does automated unit test generation really help software testers? A controlled empirical study. *ACM Trans. Softw. Eng. Methodol.* 24, 4 (2015), 1–49.
[34] Andrew Gelman and Jennifer Hill. 2006. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press.
[35] J. Ginsparg. 1978. Natural language processing in an automatic programming domain.
[36] Shuchi Grover, S. Basu, and Patricia K. Schank. 2018. What we can learn about student learning from open-ended programming projects in middle school computer science. In *49th ACM Technical Symposium on Computer Science Education*.
[37] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.
[38] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM SIGPLAN Not.* 46, 1 (2011), 317–330.
[39] Sonia Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. Lucia, and T. Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *35th International Conference on Software Engineering (ICSE)*. 842–851.
[40] Tatsunori B. Hashimoto, Kelvin Guu, Yonatan Oren, and Percy S. Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. In *Conference on Advances in Neural Information Processing Systems (NeurIPS)*. 10052–10062.
[41] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 925–930. DOI: https://doi.org/10.18653/v1/D18-1111
[42] Andrew Head, Elena Leah Glassman, B. Hartmann, and Marti A. Hearst. 2018. Interactive extraction of examples from existing code. In *CHI Conference on Human Factors in Computing Systems*.
[43] Andrew Head, Elena Leah Glassman, Gustavo Soares, R. Suzuki, Lucas Figueredo, L. D'Antoni, and B. Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *4th ACM Conference on Learning @ Scale*.
[44] George E. Heidorn. 1976. Automatic programming through natural language dialogue: A survey. *IBM J. Res. Devel.* 20, 4 (1976), 302–313.
[45] E. Hill, Manuel Roldan-Vega, J. Fails, and Greg Mallet. 2014. NL-based query refinement and contextualized code search results: A user study. In *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 34–43.
[46] Joseph L. Hodges Jr. and Erich L. Lehmann. 1963. Estimates of location based on rank tests. *Ann. Math. Statist.* (1963), 598–611.
[47] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[48] Srini Iyer, Ioannis Konstas, A. Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[49] Srini Iyer, Ioannis Konstas, A. Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[50] Paul C. D. Johnson. 2014. Extension of Nakagawa & Schielzeth's R^2_{GLMM} to random slopes models. *Meth. Ecol. Evolut.* 5, 9 (2014), 944–946.

[51] Siddharth Karamcheti, Dorsa Sadigh, and Percy Liang. 2020. Learning adaptive language interfaces through decomposition. In *1st Workshop on Interactive and Executable Semantic Parsing*. Association for Computational Linguistics, 23–33. DOI: https://doi.org/10.18653/v1/2020.intexsempar-1.4

[52] I. Keivanloo, J. Rilling, and Ying Zou. 2014. Spotting working code examples. In *36th International Conference on Software Engineering (ICSE)*.

[53] Mary Beth Kery, Amber Horvath, and B. Myers. 2017. Variolite: Supporting exploratory programming by data scientists. *CHI Conference on Human Factors in Computing Systems (CHI)*.

[54] Mary Beth Kery and B. Myers. 2017. Exploring exploratory programming. In *IEEE Symposium on Visual Languages and Human-centric Computing (VL/HCC)*. 25–29.

[55] A. Ko and B. Myers. 2004. Designing the whyline: A debugging interface for asking questions about program behavior. In *CHI Conference on Human Factors in Computing Systems (CHI)*.

[56] A. Ko and B. Myers. 2008. Debugging reinvented. In *ACM/IEEE 30th International Conference on Software Engineering (ICSE)*. 301–310.

[57] Amy Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *IEEE Symposium on Visual Languages and Human-centric Computing (VL/HCC)*. IEEE, 199–206.

[58] Ned Kock and Gary Lynn. 2012. Lateral collinearity and misleading results in variance-based SEM: An illustration and recommendations. *J. Assoc. Inf. Syst.* 13, 7 (2012).

[59] S. Kulal, Panupong Pasupat, K. Chandra, Mina Lee, Oded Padon, A. Aiken, and Percy Liang. 2019. SPoC: Search-based pseudocode to code. In *33rd Conference on Neural Information Processing Systems (NeurIPS)*.

[60] Nate Kushman and R. Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (HLT-NAACL)*.

[61] Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju. 2016. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *J. Softw. Evolut. Process* 28, 7 (2016), 589–618.

[62] Vu Le and Sumit Gulwani. 2014. FlashExtract: A framework for data extraction by examples. *ACM SIGPLAN Not.* 49, 6 (2014), 542–553.

[63] Tao Lei, F. Long, R. Barzilay, and M. Rinard. 2013. From natural language specifications to program input parsers. In *51st Annual Meeting of the Association for Computational Linguistics (ACL)*.

[64] Toby Jia-Jun Li, Amos Azaria, and B. Myers. 2017. SUGILITE: Creating multimodal smartphone automation by demonstration. In *CHI Conference on Human Factors in Computing Systems (CHI)*.

[65] Toby Jia-Jun Li, I. Labutov, X. Li, X. Zhang, W. Shi, Wanling Ding, Tom Michael Mitchell, and B. Myers. 2018. APPINITE: A multi-modal interface for specifying data descriptions in programming by demonstration using natural language instructions. In *IEEE Symposium on Visual Languages and Human-centric Computing (VL/HCC)*. 105–114.

[66] Toby Jia-Jun Li, Marissa Radensky, J. Jia, Kirielle Singarajah, Tom Michael Mitchell, and B. Myers. 2019. PUMICE: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *32nd Annual ACM Symposium on User Interface Software and Technology (UIST)*.

[67] H. Lieberman, F. Paternò, Markus Klann, and V. Wulf. 2006. End-user development: An emerging paradigm. In *End User Development*.

[68] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior. 2016. Latent predictor networks for code generation. In *54th Annual Meeting of the Association for Computational Linguistics (ACL)*. The Association for Computer Linguistics. DOI: https://doi.org/10.18653/v1/p16-1057

[69] C. Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and J. Grundy. 2020. Opportunities and challenges in code search tools. *ArXiv* abs/2011.02297 (2020).

[70] X. Liu, Beijun Shen, H. Zhong, and Jiangang Zhu. 2016. EXPSOL: Recommending online threads for exception-related bug reports. In *23rd Asia-Pacific Software Engineering Conference (APSEC)*. 25–32.

[71] Meili Lu, Xiaobing Sun, S. Wang, D. Lo, and Yucong Duan. 2015. Query expansion via WordNet for effective code search. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 545–549.

[72] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Trans. Comput. Educ.* 10 (2010), 16:1–16:15.

[73] Christopher D. Manning, Hinrich Schütze, and Prabhakar Raghavan. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

[74] Mehdi Manshadi, Daniel Gildea, and James F. Allen. 2013. Integrating programming by example and natural language programming. In *AAAI Conference on Artificial Intelligence (AAAI)*.
[75] T. McCabe. 1976. A complexity measure. *IEEE Trans. Softw. Eng.* SE-2 (1976), 308–320.
[76] Rada Mihalcea, Hugo Liu, and Henry Lieberman. 2006. NLP (natural language processing) for NLP (natural language programming). In *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer, 319–330.
[77] Parastoo Mohagheghi and Reidar Conradi. 2007. Quality, productivity and economic benefits of software reuse: A review of industrial studies. *Empir. Softw. Eng.* 12, 5 (2007), 471–516.
[78] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How can I use this method? In *IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 880–890.
[79] Yair Mundlak. 1978. On the pooling of time series and cross section data. *Economet.: J. Economet. Societ.* (1978), 69–85.
[80] Lauren Murphy, Mary Beth Kery, Oluwatosin Alliyu, Andrew Macvean, and Brad A. Myers. 2018. API designers in the field: Design practices and challenges for creating usable APIs. In *IEEE Symposium on Visual Languages and Human-centric Computing (VL/HCC)*. IEEE, 249–258.
[81] B. Myers, J. Pane, and A. Ko. 2004. Natural programming languages and environments. *Commun. ACM* 47 (2004), 47–52.
[82] B. Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59 (2016), 62–69.
[83] Brad A. Myers, Amy Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52.
[84] Brad A. Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69.
[85] Shinichi Nakagawa and Holger Schielzeth. 2013. A general and simple method for obtaining R2 from generalized linear mixed-effects models. *Meth. Ecol. Evolut.* 4, 2 (2013), 133–142.
[86] Daye Nam, Amber Horvath, Andrew Macvean, Brad Myers, and Bogdan Vasilescu. 2019. Marble: Mining for boilerplate code to identify API usability problems. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 615–627.
[87] T. Nguyen and C. Csallner. 2015. Reverse engineering mobile application user interfaces with REMAUI (T). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 248–259.
[88] Lorelli S. Nowell, Jill M. Norris, Deborah E. White, and Nancy J. Moules. 2017. Thematic analysis: Striving to meet the trustworthiness criteria. *Int. J. Qualit. Meth.* 16, 1 (2017), 1609406917733847.
[89] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A method for automatic evaluation of machine translation. In *40th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 311–318. DOI: https://doi.org/10.3115/1073083.1073135
[90] Emilio Parisotto, Abdel Rahman Mohamed, R. Singh, L. Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-symbolic program synthesis. In *5th International Conference on Learning Representations (ICLR)*.
[91] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Seahawk: Stack overflow in the IDE. In *International Conference on Software Engineering (ICSE)*. IEEE, 1295–1298.
[92] Luca Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. 2014. Mining Stack Overflow to turn the IDE into a self-confident programming prompter. In *International Conference on Mining Software Repositories (MSR)*.
[93] David Price, Ellen Rilofff, Joseph Zachary, and Brandon Harvey. 2000. NaturalJava: A natural language interface for programming in Java. In *International Conference on Intelligent User Interfaces (IUI)*. 207–211.
[94] Sebastian Proksch, Sven Amann, and Sarah Nadi. 2018. Enriched event streams: A general dataset for empirical studies on in-IDE activities of software developers. In *15th International Conference on Mining Software Repositories (MSR)*. 62–65.
[95] Karthik Radhakrishnan, Arvind Srikantan, and Xi Victoria Lin. 2020. ColloQL: Robust Text-to-SQL over search queries. In *1st Workshop on Interactive and Executable Semantic Parsing*. 34–45.
[96] Mukund Raghothaman, Y. Wei, and Y. Hamadi. 2016. SWIM: Synthesizing what I mean—Code search and idiomatic snippet synthesis. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 357–367.
[97] M. M. Rahman and C. Roy. 2014. SurfClipse: Context-aware meta-search in the IDE. In *IEEE International Conference on Software Maintenance and Evolution*. 617–620.
[98] Mohammad Masudur Rahman, Shamima Yeasmin, and Chanchal K. Roy. 2014. Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 194–203.
[99] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM, 419–428.
[100] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional program synthesis from natural language and examples. In *24th International Joint Conference on Artificial Intelligence (IJCAI)*.

[101] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.
[102] Matthew Richardson, Ewa Dominowska, and Robert Ragno. 2007. Predicting clicks: Estimating the click-through rate for new ads. In *16th International Conference on World Wide Web*. 521–530.
[103] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2020. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 287–298.
[104] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How developers search for code: A case study. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 191–201.
[105] Apurvanand Sahay, Arsene Indamutsa, D. D. Ruscio, and A. Pierantonio. 2020. Supporting the understanding and comparison of low-code development platforms. In *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 171–178.
[106] Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. In *33rd Conference on Neural Information Processing Systems (NeurIPS)*.
[107] Forrest Shull, Janice Singer, and Dag I. K. Sjøberg. 2007. *Guide to Advanced Empirical Software Engineering*. Springer.
[108] Armando Solar-Lezama. 2008. Program synthesis by sketching.
[109] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *International Conference on Software Engineering (ICSE)*.
[110] Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. 2019. Does BLEU score work for code migration? In *IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 165–176.
[111] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 269–280.
[112] David Vadas and James R. Curran. 2005. Programming with unrestricted natural language. In *Australasian Language Technology Workshop*. 191–199.
[113] Venkatesh Vinayakarao, A. Sarma, R. Purandare, Shuktika Jain, and Saumya Jain. 2017. ANNE: Improving source code search using entity retrieval approach. In *Web Search and Data Mining Conference (WSDM)*.
[114] Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*. 1332–1342.
[115] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. 2015. *Building Bing Developer Assistant*. Technical Report. MSR-TR-2015-36, Microsoft Research.
[116] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Science & Business Media.
[117] Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 6045–6052.
[118] Xuchen Yao and Benjamin Van Durme. 2014. Information extraction over structured data: Question answering with freebase. In *52nd Annual Meeting of the Association for Computational Linguistics (ACL)*. 956–966.
[119] Ziyu Yao, Xiujun Li, Jianfeng Gao, Brian Sadler, and Huan Sun. 2019. Interactive semantic parsing for if-then recipes via hierarchical reinforcement learning. In *AAAI Conference on Artificial Intelligence (AAAI)*. 2547–2554.
[120] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. CoaCor: Code annotation for code retrieval with reinforcement learning. In *World Wide Web Conference (WWW)*.
[121] Ziyu Yao, Daniel S. Weld, W. Chen, and Huan Sun. 2018. StaQC: A systematically mined question-code dataset from stack overflow. In *World Wide Web Conference (WWW)*.
[122] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories (MSR)*. ACM, 476–486. DOI: https://doi.org/10.1145/3196398.3196408
[123] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.
[124] Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP), Demo Track*.
[125] Pengcheng Yin and Graham Neubig. 2019. Reranking for neural semantic parsing. In *57th Annual Meeting of the Association for Computational Linguistics (ACL)*. Association for Computational Linguistics, 4553–4559. DOI: https://doi.org/10.18653/v1/P19-1447
[126] Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. 2018. NAPS: Natural program synthesis dataset. In *2nd Workshop on Neural Abstract Machines & Program Induction (NAMPI)*.

[127] John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *National Conference on Artificial Intelligence*. 1050–1055.

[128] Luke Zettlemoyer and Michael Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. 678–687.

[129] Ruiqi Zhong, Mitchell Stern, and D. Klein. 2020. Semantic scaffolds for pseudocode-to-code generation. In *Meeting of the Association for Computational Linguistics*.

[130] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).

Received January 2021; revised July 2021; accepted September 2021