CREATING CONTEXTUALIZED CODE SNIPPETS FROM NATURAL LANGUAGE INTENTS IN VSCODE.

Anonymous authors
Paper under double-blind review

ABSTRACT

During software development, developers often turn to code snippets as a starting point for their individual use and further development. These code snippets can be looked up and explored on popular Q&A websites such as Stack Overflow for solving programming-related problems. The context switch between the IDE and the respective Q&A websites required for this affects the programming flow and degrades the developer's programming productivity. We present an extension for the VSCode IDE to reduce the need for context switches. Multiple code generation models built on GPT-3 or Codex can be plugged in and used to generate and contextualize code snippets based on a natural language intent. A qualitative test and a brief pricing analysis indicate that our open-source extension, especially when combined with Codex, can already be a valuable addition to everyday software development and, furthermore, enables developers to benefit from future improvements in natural language processing.

1 INTRODUCTION

The internet provides developers a great deal of programming help, most notably through the popular Q&A website Stack Overflow. Today, this wealth of source code data is used to train large neural networks that enable code generation based on natural language intent (Chen et al., 2021; Feng et al., 2020). It has become standard practice for developers to search specifically for code snippets to solve programming-related problems. The search query is usually formulated as an intention suitable for search engines—often in natural language—to be executed by general purpose search engines such as Google or Bing. From these, the programmer eventually selects a code snippet, which is then copied, pasted into the IDE, sometimes ported, integrated and customized, and further developed. However, research, e.g., by Bacchelli et al. (2012), attest to the negative impact of using websites like Stack Overflow during programming. The major weaknesses are (1) the context switch when leaving the IDE to open a browser and (2) the lack of context of copied code snippets from the web. Our extension aims to work around these adverse side effects by using state-of-the-art code generation models to provide contextualized code suggestions directly in the IDE.

The main contributions of this work are

1. a VSCode extension that suggests code snippets based on natural language intents,
2. the utilization of large language models to contextualize and generate code suggestions,
3. an evaluation of different models used for the code generation, and
4. an extensible architecture to integrate additional code generation models to the tool.

Our VSCode extension supports writing Python code and is designed to help developers focus and increase their productivity. All currently integrated models support only Python code generation. However, some of them, such as those described in chapters 4.3–4.4, are easily extensible to other programming languages.

The paper is organized as follows. Section 2 presents the related work. Section 3 introduces the extension itself and highlights its implementation. Thereafter, section 4 presents the currently available code generation models of the extension with a focus on the implementation details. Building on this, section 5 compares the models and section 6 summarizes this work.

2 RELATED WORK

Recent work focused on utilizing large amounts of available source code online to improve deep-learning-based code generations models (Chen et al., 2021; Feng et al., 2020; Svyatkovskiy et al., 2020; Xu et al., 2020; Kanade et al., 2020; Phan et al., 2021). These models are built on transformer-based architectures such as BERT (Feng et al., 2020; Kanade et al., 2020), GPT (Chen et al., 2021; Svyatkovskiy et al., 2020), or T5 (Phan et al., 2021). Other work utilized these models within IDEs for searching or generating code snippets (Zhang et al., 2016; Xu et al., 2021; Ponzanelli et al., 2014). In addition, benchmarks for code generation models have been established to allow for an evaluation of different models (Austin et al., 2021; Lu et al., 2021). The following two papers are the primary resources on whose findings our work expands.

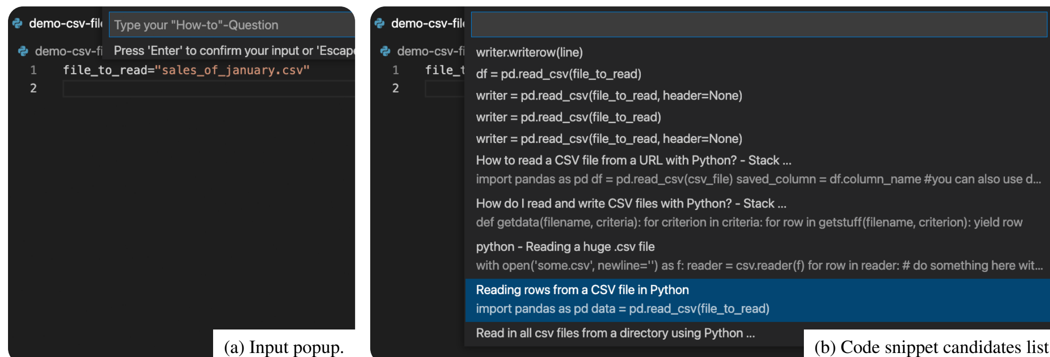
In-IDE Code Generation from Natural Language Xu et al. (2021) developed a PyCharm IDE plugin that implements a hybrid of code generation and code retrieval functionality. Based on a given natural language intent, the extension creates seven different code snippets with a code generation model and retrieves seven matching code snippets from Stack Overflow. The authors analyzed the impact of their plugin on developer productivity through a study with 31 participants. Fourteen Python programming tasks were created for seven different use cases, such as file manipulation, web scraping, or data visualization. The results showed that they did not find any significant performance increase while the developers used the plugin. Code generated or retrieved by the plugin was neither faster written nor of higher quality. Nevertheless, they concluded their work with six recommendations for future work that might increase developer productivity through their plugin. One of the recommendations that this work focuses on is considering a developer's local context inside the current file as part of the input and output of the code generation model.

CoPilot using Codex Chen et al. (2021) introduced the Codex model; a large language model fine-tuned on code from GitHub. It can be accessed as a VSCode Extension called *GitHub CoPilot*[1], released in June 2021. The extension provides a similar user experience of retrieving inline code snippet suggestions based on a natural language intent. However, this extension does not allow one to choose one's preferred code generation model based on considerations such as accuracy or cost. Additionally, CoPilot is not open-source, making it difficult to understand what parts of the source code are sent to the Microsoft servers. We designed our Extensions to provide a generalized workflow for code snippet suggestions generated using models chosen by the developer.

3 THE EXTENSION

The developed VSCode extension generates code snippets directly in the IDE. It takes a query as input and returns a list of code snippets that match the query via a customizable code generation model. There are two ways to interact with the extension.

[1] copilot.github.com



(a) Input popup. (b) Code snippet candidates list.

Figure 1: Query-Approach: A query is taken as input and code snippet candidates are presented in a list, generated by the default code generation+retrieval model and the contextualization using GPT-3.

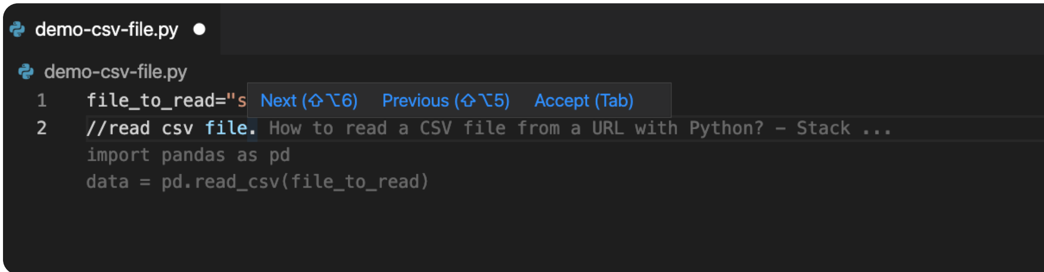


Figure 2: Inline-Approach: The intent `//read csv file.` is triggering the inline suggestions and key commands to switch between code snippet suggestions.

The first option, the *query-approach*, similar to the work of Xu et al. (2021), utilizes the Quick Pick capability[2] of the VSCode API to collect the user input and then shows a list of code snippet candidates (see Figure 1). After selecting a snippet, it is pasted into the IDE. This option is accessible via a keyboard shortcut which opens a popup where the user can enter a request in natural language (see Figure 1 (a)). The plugin then sends the request to the code generation model and displays the list of code snippet candidates shown in Figure 1 (b).

The second option, the *inline-approach*, features a similar user experience to GitHub CoPilot. It uses the Inline Suggestions capabilities[3] of the VSCode API to show the code snippet candidates directly as if they were already accepted inside of the IDE (see Figure 2). The extension's inline suggestion capabilities can be triggered by adding two slashes followed by the query and ending with a dot.

An additional feature of the extension is that the user can choose the code generation model. The code generations models introduced in the next chapter have their unique benefits and trade-offs. In the "Settings" tab of the VSCode extension, the different models can be selected if the required access keys for the models have been provided (see Figure 3). Additionally, the repository provides instructions on plugging in custom code generation models.

4 Code Generation Models

For our extension, we integrated (1) a default code generation and retrieval based on *Stack Overflow* and the *Bing Search Engine*, (2) a contextualization and a (3) a code generation using *GPT-3* as well as (4) contextualized code generation using *Codex*.

[2]*Quick Pick API:* code.visualstudio.com/api/extension-capabilities/common-capabilities#quick-pick
[3]*Inline Suggestions API:* code.visualstudio.com/updates/v1_58#_inline-suggestions

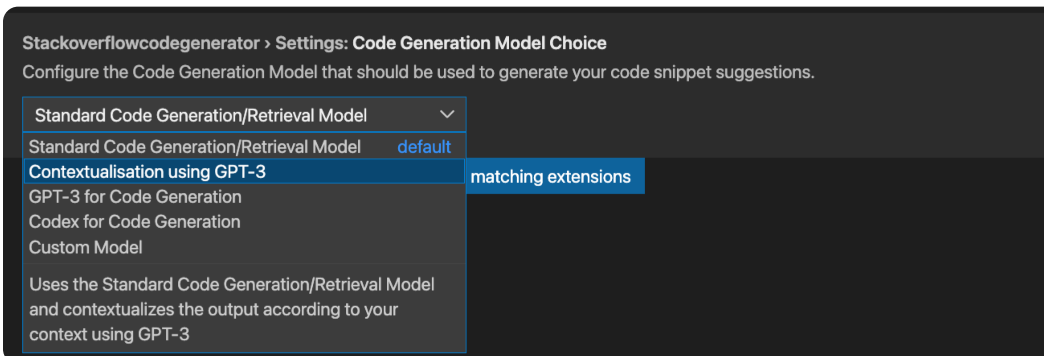


Figure 3: Extension settings to choose the preferred Code Generation Model.

4.1 DEFAULT CODE GENERATION+RETRIEVAL MODEL

The default model of the extension uses a similar architecture to the one Xu et al. (2021) used in their PyCharm plugin. The code generation model is based on a previous paper of the same authors Xu et al. (2020). First, they used a pre-trained model based on a dataset of 100k Stack Overflow Code-Question pairs for the programming language Python and API Documentations. Then, they fine-tuned the model on a small manually curated dataset called *CoNaLa*[4] of code-question pairs that had a significantly higher data quality.

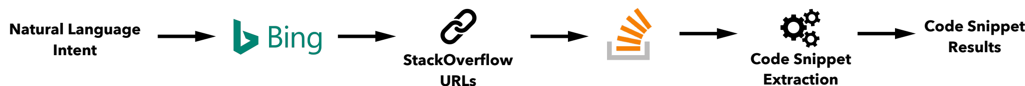


Figure 4: Code Retrieval Pipeline

The code retrieval pipeline (see Figure 4 for a visualization of the pipeline) uses a wrapper on top of the Bing Search Engine that already includes an advanced indexing and ranking mechanism. The Bing API is called with the additional string "site:stackoverflow.com" to ensure that only the best matching Stack Overflow pages are considered in the response. The response then provides a Stack Overflow URL to the relevant posts, which can be scraped. Then, the answer with the most votes is selected, and based on a simple heuristic, a code snippet is extracted from that answer. This results in a list of code snippets that match the query. Our extension shows five results for each of the models and combines them in the output.

4.2 CONTEXTUALIZATION USING GPT-3

This option uses the default code generation+retrieval model and then utilizes the language model GPT-3 to contextualize the code snippets to fit the context of the current position in the file. An option to contextualize a code snippet is to replace the variables with the matching ones of the current context of a file.

GPT-3 is an autoregressive language model with 175 billion parameters released by OpenAI in July 2020 (Brown et al., 2020). Through a technique called few-shot learning, GPT-3 can be taught to solve various tasks. The only thing necessary is to adjust the so-called prompt, i.e., the input to the neural network. For the design of this prompt, it is helpful first to describe the task of the model and list some examples for it in the next step.

The prompt for the model to contextualize code snippets starts with the formulation of the task:

```
1 | Replace variables in Python Code Snippet (Unprocessed) with the context of the Program:
```

To increase the model's accuracy on this task, we provide a few examples to the prompt on how a given unprocessed code snippet should be contextualized given the context of its program. The examples start with adding the context of the position in a current file (`Program:`), then adding the unprocessed code snippet generated by the default code generation/retrieval model (`Unprocessed:`), and finally adding the contextualized code snippet (`Processed:`). An example taken from the prompt is the following:

```
1 | Program:
2 | fruits = ["apple", "banana", "cherry"]
3 | new_fruit = "mango"
4 | Unprocessed:
5 | list.append(2)
6 | Processed:
7 | fruits.append(new_fruit)
```

By providing GPT-3 with examples like this, the model can be taught to only substitute variables to the matching type. The complete prompt with all examples can be found in Appendix A.1.

[4] *CoNaLa:* conala-corpus.github.io

4.3 CODE GENERATION USING GPT-3

The following model option generates the code snippet and contextualizes the output via one prompt to GPT-3. It removes the need to use the default code generation and retrieval model for receiving a list of code snippets for a natural language intent.

GPT-3 was not explicitly built for generating source code, but it was trained on a scrape of the whole internet, which includes websites with source code such as GitHub or Stack Overflow (Brown et al., 2020, p.3). This makes it possible to tweak GPT-3 to output code snippets instead of natural language. The created prompt for instructing GPT-3 to create Python code snippets from a natural language intent starts with the following task definition:

```
CoPilot is a chatbot that helps software developers write code.
It takes a natural language intent NLI and answers in the Python programming language.
```

Afterwards, the model is given some examples that consist of the context of a program (`Program:`), the question or task instruction as a natural language intent (`NLI:`) and the matching code snippet output (`CoPilot:`). An example looks like this:

```
Program:
fruits = ["apple", "banana", "cherry"]
new_fruit = "mango"
NLI:
Append new fruit to list of fruits
CoPilot:
fruits.append(new_fruit)
```

The complete prompt with all examples can be found in Appendix A.2.

4.4 CODEX FOR CODE GENERATION

Codex[5] is a model that was released by OpenAI through an API in private beta in August 2021. They fine-tuned a GPT model on code from GitHub. The performance for generating code improved significantly compared to code generation using GPT-3 (Chen et al., 2021).

The model can create code in over a dozen programming languages, including JavaScript, Go, PHP, and Python. Since the extension focuses on generating Python code snippets, the only thing that needs to be specified in the prompt is to output Python code. This is as easy as adding the following line to the beginning of the prompt:

```
# Python 3
```

After this line, it is possible to add the current context of the file the developer is working in and the natural language intent to the prompt, at which point the Codex model will respond with a matching code snippet.

5 RESULTS

We evaluated the different code generation models by means of qualitative testing and pricing.

Qualitative Testing In general, the default code generation and retrieval model introduced by Xu et al. (2021) has a relatively low quality. If the extension creates ten code snippet results for a query, usually only a few of them are useful (see Appendix B.1 for examples of the model).

The low quality of the code snippets created by the default model, in turn, harms the second model, which builds on top of it and only does the contextualization using GPT-3.

Using GPT-3 for code generation works great for queries that are similar to the ones used as examples in the prompt, but it often doesn't find any useful code snippets for more advanced queries (see Appendix B.3 for examples).

By far the best performing model during the qualitative testing was the Codex model. Even though only minor modifications were necessary for the prompt, the generated code snippets matched the

[5] *Codex Release:* openai.com/blog/openai-codex

queries best. Unlike the default code generation model, for simple queries most of the code snippets created by Codex are usable and compilable.

Pricing The Codex API is currently offered for free in private beta. Therefore, we cannot include it in the price comparison.

The default code generation model was deployed on an AWS EC2 server. Also, the code retrieval pipeline does not add additional costs for generating a list of code snippet candidates. For these reasons, this model represents the long-term least expensive option.

Next, the two GPT-3 based models are examined for their costs. OpenAI offers different model sizes for their GPT-3 language model. The larger ones have a better performance, but cost significantly more. Additionally, the pricing depends on the length of the prompt and the generated code snippet. The price is paid in 1,000-token increments, with the number of tokens corresponding to the byte-pair encoded input and output of the model. On average, 1,000 tokens correspond to approximately 750 words[6]. However, more tokens must be expected for code, as their byte-pair encoding algorithm is not optimal for encoding source code.

As described in Section 4.2, one of the models uses GPT-3 just for the contextualization of the code snippets that it retrieves from the default code generation/retrieval model and the other model uses GPT-3 directly to generate the code snippets. Since the evaluation of the default code generation/retrieval model has shown that a larger amount of possible code snippet candidates must be created with this model in order to obtain at least some useful results, the GPT-3 API must be called more often for the model that uses GPT-3 for the contextualization. For a single use of the extension, the GPT-3 for contextualization model makes 10 request to the GPT-3 API, while the GPT-3 for code generation model makes just one request and generates 3 code snippets.

The prompts for both of the models have a similar length because they consist of the same number of examples. This makes using the GPT-3 for contextualization model about three times more expensive than using the GPT-3 for code generation model. An average single use of the extension with the second largest GPT-3 model costs about $0.06 (~10,000 tokens) for the GPT-3 for contextualization model and $0.02 (~3,000 tokens) for the GPT-3 for code generation model. Using the largest GPT-3 model would increase the price to about $0.6 for the GPT-3 for contextualization model and $0.2 for the GPT-3 for code generation model. Although the performance of both models increases when using the larger model, regular use of the extension with this configuration would not be economically feasible. Therefore, we decided to set the second largest GPT-3 model as the default, which still provides good performance but reduces the cost significantly. See Appendix C for a detailed price calculation for the different models.

6 CONCLUSION

In this work, we introduced a VSCode extension that provides code suggestions based on a customizable code generation model. Developers can submit a query to one of the code generation models and retrieve a list of code snippet suggestions from which to choose. We also introduced the four code generation models currently available in the extension, i.e., the standard code generation and query model, the contextualization model using GPT-3, the code generation model using GPT-3, and the codex model. We believe that code generation models will become even more potent in the future and will be an indispensable tool for any developer. Our extension provides a flexible and extensible architecture that could serve as a foundation for future work in this direction.

Our extension is open-source and freely available online at github.com/blinded/for-review.

ACKNOWLEDGMENTS

Blinded for review.

[6] *GPT-3 Pricing:* beta.openai.com/pricing

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

Alberto Bacchelli, Luca Ponzanelli, and Michele Lanza. Harnessing stack overflow for the ide. In *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, RSSE '12, pp. 26–30. IEEE Press, 2012. ISBN 9781467317597.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Pre-trained contextual embedding of source code. *CoRR*, abs/2001.00059, 2020. URL http://arxiv.org/abs/2001.00059.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021. URL https://arxiv.org/abs/2102.04664.

Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. CoTexT: Multi-task learning with code-text transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pp. 40–47, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.nlp4prog-1.5. URL https://aclanthology.org/2021.nlp4prog-1.5.

Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. 05 2014. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597077.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pp. 1433–1443, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3417058. URL https://doi.org/10.1145/3368089.3417058.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 6045–6052, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.538. URL https://aclanthology.org/2020.acl-main.538.

Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. In-ide code generation from natural language: Promise and challenges. *CoRR*, abs/2101.11149, 2021. URL https://arxiv.org/abs/2101.11149.

Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. Bing developer assistant: Improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pp. 956–961, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2983955. URL https://doi.org/10.1145/2950290.2983955.

A GPT-3 PROMPTS

A.1 CONTEXTUALIZATION USING GPT-3

```
Replace variables in Python Code Snippet (Unprocessed) with
the context of the Program:

Program:
file_to_download="sales_of_january.csv"
Unprocessed:
import pandas as pd
data = pd.read_csv('https://example.com/passkey=wedsmdjsjmdd')
Processed:
import pandas as pd
data = pd.read_csv(file_to_download)

Program:
fruits = ["apple", "banana", "cherry"]
new_fruit = "mango"
Unprocessed:
list.append(2)
Processed:
fruits.append(new_fruit)

Program:
hostUrl = "http://localhost:8081/parse/conala"
parameters = {
    q: query
};
Unprocessed:
requests.get(url, params=params)
Processed:
requests.get(hostUrl, params=parameters)

Program:
params = {
    q: query
};
randomUrl = "https://example.com"
Unprocessed:
requests.get(url, params=parameters)
Processed:
requests.get(randomUrl, params=parameters)

Program:
d1 = {'a': 1, 'b': 2}
```

```
d2 = {'b': 10, 'c': 11}
Unprocessed:
z = x | y
Processed:
z = d1 | d2

Program:
prompt_for_email = "Please enter your email"
Unprocessed:
text = input("prompt")
Processed:
text = input(prompt_for_email)

Program:
file_to_open = "earnings_Q2_20.xlsm"
Unprocessed:
import os.path
os.path.isfile(fname)
Processed:
import os.path
os.path.isfile(file_to_open)

Program:
colors = ['red', 'blue', 'green']
favoriteColor = 'red'
Unprocessed:
list[-1]
Processed:
colors[-1]

Program:
new_list = list()
Unprocessed:
if len(li) == 0:
    print('the list is empty')
Processed:
if len(new_list) == 0:
    print('the list is empty')

Program:
old_list = [1,2,3,4,5,6]
Unprocessed:
lst2=lst1[:]
Processed:
new_list = old_list[:]

Program:
list1 = [23,65,23,75,23]
list2 = [245,95,122,1,98]
Unprocessed:
c = [x+y for x,y in zip(a, b)]
Processed:
sum_of_lists = [x+y for x,y in zip(list1, list2)]
```

Listing 1: Complete prompt for contextualizing code snippets with the help of GPT-3

A.2 GPT-3 FOR CODE GENERATION

```
CoPilot is a chatbot that helps software developers write code.
It takes a natural language intent NLI and answers in the
Python programming language.

Program:
file_to_download="sales_of_january.csv"
NLI:
Read the data of the csv file
CoPilot:
import pandas as pd
data = pd.read_csv(file_to_download)

Program:
fruits = ["apple", "banana", "cherry"]
new_fruit = "mango"
NLI:
Append new fruit to list of fruits
CoPilot:
fruits.append(new_fruit)

Program:
const hostUrl = "http://localhost:8081/parse/conala"
```

```
23  const parameters = {
24      q: query
25  };
26  NLI:
27  Make a get request with parameters
28  CoPilot:
29  requests.get(hostUrl, params=parameters)

31  Program:
32  const params = {
33      q: query
34  };
35  const randomUrl = "https://example.com"
36  NLI:
37  Make a get request with parameters
38  CoPilot:
39  import requests
40  requests.get(randomUrl, params=parameters)

42  Program:
43  csvFile = "./data.csv"
44  NLI:
45  Read the data of the csv file
46  CoPilot:
47  import pandas as pd
48  data = pd.read_csv(csvFile)

50  Program:
51  d1 = {'a': 1, 'b': 2}
52  d2 = {'b': 10, 'c': 11}
53  NLI:
54  Merge two dictionaries
55  CoPilot:
56  z = d1 | d2

58  Program:
59  prompt_for_email = "Please enter your email"
60  NLI:
61  Request user input from command line
62  CoPilot:
63  text = input(prompt_for_email)

65  Program:
66  file_to_open = "earnings_Q2_20.xlsm"
67  NLI:
68  Check if file exists
69  CoPilot:
70  import os.path
71  os.path.isfile(file_to_open)

73  Program:
74  colors = ['red', 'blue', 'green']
75  favoriteColor = 'red'
76  NLI:
77  Get last item of list
78  CoPilot:
79  colors[-1]

81  Program:
82  new_list = list()
83  NLI:
84  Check if list is empty
85  CoPilot:
86  if len(new_list) == 0:
87      print('the list is empty')

89  Program:
90  old_list = [1,2,3,4,5,6]
91  NLI:
92  Clone list
93  CoPilot:
94  new_list = old_list[:]

96  Program:
97  list1 = [23,65,23,75,23]
98  list2 = [245,95,122,1,98]
99  NLI:
100 Sum elements in two lists
101 CoPilot:
102 sum_of_lists = [x+y for x,y in zip(list1, list2)]
```

```
Program:
cars = ["Ford", "Volvo", "BMW"]
NLI:
Sort list
CoPilot:
cars.sort()

Program:
my_list = [1,1,3,5,5]
NLI:
Find duplicate in list
CoPilot:
duplicates = [x for x in my_list if x in my_list]

Program:
name = "Mamma"
NLI:
Check if palindrome
CoPilot:
palindrome = name == name[::-1]
```

Listing 2: Complete prompt for generating code snippets with the help of GPT-3

B CODE COMPLETION EXAMPLES

Example 1 - List Operations
Context of the file:

```
l_1 = ["test", 2, 3.5]
l_2 = [6, "test2", 9.7]
```

Query:

```
//merge two lists.
```

Example 2 - File Manipulation
Context of the file:

```
file_to_write_to = "./transcript.txt"
text_to_add = "Hello World"
```

Query:

```
//add string to file.
```

B.1 DEFAULT CODE GENERATION/RETRIEVAL

```
# Disclaimer: Even though this model does not replace the variable names, the compilable
    solutions that do what the query specifies have been marked as correct.

# Example 1:

### COMPLETION 1 (WRONG -> At least not what was expected, since it aggregates list in a
    tuple): ###

zip(T, y)

### COMPLETION 2 (WRONG): ###

dict(list(x) for x in zip(*my_list))

### COMPLETION 3 (WRONG): ###

dict(list(x) for x in zip(*lists))

### COMPLETION 4 (WRONG): ###

dict(list(y) for y in zip(a, b, c))

### COMPLETION 5 (WRONG): ###

zip(T, b)

### COMPLETION 6 (CORRECT): ###

listone = [1, 2, 3]
listtwo = [4, 5, 6]

joinedlist = listone + listtwo
>>> joinedlist
[1, 2, 3, 4, 5, 6]


### COMPLETION 7 (CORRECT): ###

>>> l1 = [1, 2, 3]
>>> l2 = [4, 5, 6]
>>> joined_list = [*l1, *l2] # unpack both iterables in a list literal
>>> print(joined_list)
[1, 2, 3, 4, 5, 6]
l = [1, 2, 3]
r = range(4, 7)
res = l + r
res = [*l, *r]


### COMPLETION 8 (CORRECT): ###

import itertools
for item in itertools.chain(listone, listtwo):
   # Do something with each list item


### COMPLETION 9 (CORRECT): ###

mergedlist = list(set(listone + listtwo))
```

```
### COMPLETION 10 (CORRECT): ###

listone = [1,2,3]
listtwo = [4,5,6]

listone.extend(listtwo)
mergedlist = []
mergedlist.extend(listone)
mergedlist.extend(listtwo)


# Example 2:

### COMPLETION 1 (CORRECT): ###

f.write('en_US')

### COMPLETION 2 (WRONG): ###

os.write(string)

### COMPLETION 3 (WRONG): ###

os.write(file)

### COMPLETION 4 (WRONG): ###

os.listdir('..')

### COMPLETION 5 (WRONG): ###

f.write('\n'.join(line.rstrip('\n') for line in open('file.txt')))

### COMPLETION 6 (CORRECT): ###

with open("Output.txt", "w") as text_file:
    text_file.write("Purchase Amount: %s" % TotalAmount)
text_file = open("Output.txt", "w")
text_file.write("Purchase Amount: %s" % TotalAmount)
text_file.close()
with open("Output.txt", "w") as text_file:
    text_file.write("Purchase Amount: {0}".format(TotalAmount))
with open("Output.txt", "w") as text_file:
    print("Purchase Amount: {}".format(TotalAmount), file=text_file)
with open("Output.txt", "w") as text_file:
    print(f"Purchase Amount: {TotalAmount}", file=text_file)

### COMPLETION 7 (CORRECT): ###

price = 33.3
with open("Output.txt", "w") as text_file:
    text_file.write("Purchase Amount: %s price %f" % (TotalAmount, price))

### COMPLETION 8 (WRONG): ###

f = open('%s.csv' % name, 'wb')

### COMPLETION 9 (CORRECT - but a lot of stuff we didn't ask for in the query): ###

your_data = {"Purchase Amount": 'TotalAmount'}
print(your_data, file=open('D:\log.txt', 'w'))
def my_func():
    """
     this function return some value
     :return:
     """
    return 25.256


def write_file(data):
    """
     this function write data to file
     :param data:
     :return:
     """
    file_name = r'D:\log.txt'
    with open(file_name, 'w') as x_file:
        x_file.write('{} TotalAmount'.format(data))
```

```
def run():
    data = my_func()
    write_file(data)


run()

### COMPLETION 10 (WRONG): ###

numpy.savetxt('Output.txt', ["Purchase Amount: %s" % TotalAmount], fmt='%s')
```

Listing 3: Completion options for Example 1 and 2 of the Default Code Generation/Retrieval model

B.2 CONTEXTUALIZATION USING GPT-3

```
# Disclaimer: All completions are marked as wrong if the variables weren't replaced with
    the ones from the context of the file.

# Example 1:

### COMPLETION 1 (WRONG): ###

zip(l_1, l_2)

### COMPLETION 2 (WRONG): ###

dict(list(x) for x in zip(l_1, l_2))

### COMPLETION 3 (WRONG): ###

dict(list(x) for x in zip(*lists))

### COMPLETION 4 (WRONG): ###

dict(list(y) for y in zip(l_1, l_2, l_3

### COMPLETION 5 (WRONG): ###

zip(l_1, l_2)

### COMPLETION 6 (WRONG): ###

listone = [1, 2, 3]
listtwo = [4, 5, 6]
joinedlist = listone + listtwo
>>> joinedlist
[1, 2, 3, 4, 5, 6]

### COMPLETION 7 (WRONG): ###

>>> l1 = [1, 2, 3]
>>> l2 = [4, 5, 6]
>>> joined_list = [*l1, *l2] # unpack both iterables in a list literal
>>> print(joined_list)
[1, 2, 3, 4, 5, 6]
l = [1, 2, 3]
r = range(4, 7)
res = l + r
res = [*l, *r]

### COMPLETION 8 (CORRECT): ###

import itertools
for item in itertools.chain(l_1, l_2):
    # Do something with each list item

### COMPLETION 9 (WRONG): ###

mergedlist = list(set(listone + listtwo))

### COMPLETION 10 (WRONG): ###

listone = [1,2,3]
listtwo = [4,5,6]

listone.extend(listtwo)
```

```
mergedlist = []
mergedlist.extend(listone)
mergedlist.extend(listtwo)

# Example 2:

### COMPLETION 1 (WRONG): ###

open('hello', 'w').write(text_to_add)

### COMPLETION 2 (WRONG): ###

f.write('<unk>' % file_to_write_to)

### COMPLETION 3 (WRONG): ###

f.write('My <unk>')

### COMPLETION 4 (WRONG): ###

f.write('<unk>Hello World</unk>')

### COMPLETION 5 (WRONG): ###

os.write(file_to_write_to, text_to_add)

### COMPLETION 6 (WRONG): ###

from pathlib import Path

Desktop = Path('Desktop')

# print(Desktop)
WindowsPath('Desktop')

# extend the path to include subdir
SubDeskTop = Desktop / "subdir"

# print(SubDeskTop)
WindowsPath('Desktop/subdir')
# passing an absolute path has different behavior

### COMPLETION 7 (WRONG): ###

f = open('%s.csv' % name, 'wb')

### COMPLETION 8 (WRONG): ###

file_name = "testlorem"
string_to_add = "added"

with open(file_name, 'r') as f:
    file_lines = [''.join([x.strip(), string_to_add, '\\n']) for x in f.readlines()]

with open(file_name, 'w') as f:
    f.writelines(file_lines)

### COMPLETION 9 (WRONG): ###

from pathlib import Path
Desktop = Path('Desktop')
SubDeskTop = Path.joinpath(Desktop, "subdir")

### COMPLETION 10 (WRONG): ###

with open('file.txt', 'a') as f:
    f.write('blah')
    f.writelines(lines)
```

Listing 4: Completion options for Example 1 and 2 of the GPT-3 for Contextualization model

B.3 GPT-3

```
# Example 1:

### COMPLETION 1 (CORRECT): ###
```

```
5   merged_list = l_1 + l_2

7   ### COMPLETION 2 (CORRECT): ###

9   l_1 + l_2

11  ### COMPLETION 3 (CORRECT, but generated the same as COMPLETION 2): ###

13  l_1 + l_2


16  # Example 2:

18  ### COMPLETION 1 (CORRECT): ###

20  with open(file_to_write_to, 'a') as f:
21      f.write(text_to_add)

23  ### COMPLETION 2 (CORRECT, but generated the same as COMPLETION 1): ###

25  with open(file_to_write_to, 'a') as f:
26      f.write(text_to_add)

28  ### COMPLETION 3 (CORRECT, but generated the same as COMPLETION 1 and 2): ###

30  with open(file_to_write_to, 'a') as f:
31      f.write(text_to_add)
```

Listing 5: Completion options for Example 1 and 2 of the GPT-3 for Code Generation model

B.4 CODEX

```
1   # Example 1:

3   ### COMPLETION 1 (CORRECT): ###

5   l_3 = l_1 + l_2
6   print(l_3)

8   ### COMPLETION 2 (CORRECT, but generated the same as COMPLETION 1): ###

10  l_3 = l_1 + l_2
11  print(l_3)

13  ### COMPLETION 3 (CORRECT, but generated the same as COMPLETION 1 and 2): ###

15  l_3 = l_1 + l_2
16  print(l_3)


19  # Example 2:

21  ### COMPLETION 1 (CORRECT): ###

23  f = open(file_to_write_to, "a")
24  f.write(text_to_add)
25  f.close()

27  ### COMPLETION 2 (CORRECT, but generated the same as COMPLETION 1): ###

29  file = open(file_to_write_to, "a")
30  file.write(text_to_add)
31  file.close()

33  ### COMPLETION 3 (CORRECT): ###

35  with open(file_to_write_to, "a") as f:
36      f.write(text_to_add)
```

Listing 6: Completion options for Example 1 and 2 of the Codex model

C GPT-3 PRICING EXAMPLE

	Contextualization (curie)	Contextualization (davinci)	Code Generation (curie)	Code Generation (davinci)
Number of Requests	10	10	2	2
Prompt Length (in tokens)	8797	8797	1908	1908
Completion Length (in tokens)	537	357	320	192
Total Cost	$0.06	$0.55	$0.01	$0.13

Table 1: Evaluation of a single use of the extension using the GPT-3 for Contextualization and GPT-3 for Code Generation model. Both were evaluated using the second largest GPT-3 model called ”curie” and the largest one called ”davinci”.

Context of the file:

```
def getInfos(name):
    hostUrl = "https://example.com/getInformation"
    paramsForUrl = {
        "name": name
    }
```

Query:

```
//get request with parameters.
```