PCR-Chain: Partial Code Reuse Assisted by Hierarchical Chaining of Prompts on Frozen Copilot

Qing Huang†
qh@jxnu.edu.cn
Jiangxi Normal University
China

Jiahui Zhu†
jhz@jxnu.edu.cn
Jiangxi Normal University
China

Zhilong Li
zll@jxnu.edu.cn
Jiangxi Normal University
China

Zhenchang Xing
zhenchang.xing@data61.csiro.au
CSIRO's Data61 & Australian National University
Australia

Changjing Wang*
wcj@jxnu.edu.cn
Jiangxi Normal University
China

Xiwei Xu
Xiwei.Xu@data61.csiro.au
CSIRO's Data61
Australia

***Abstract*—API documentation, technical blogs and programming Q&A sites contain a large amount of partial code that can be reused in programming tasks. However, due to unresolved simple names and last-mile syntax errors, such partial code is frequently not compilable. To facilitate partial code reuse, we develop PCR-Chain for resolving FQNs and fixing last-mile syntax errors in partial code based on a giant pre-trained code model (e.g., Copilot). Methodologically, PCR-Chain is backed up by the underlying global-level prompt architecture (which combines three design ideas: hierarchical task breakdown, prompt composition including sequential and conditional structures, and a mix of prompt-based AI and non-AI units) and the local-level prompt design. Technically, we propose PCR-Chain, which employs in-context learning rather than supervised fine-tuning with gradient updates on downstream task data. This approach enables the frozen, giant pre-trained code model to learn the desired behavior for a specific task through behavior-describing prompts and imitate it to complete the task. Experimental results show that PCR-Chain automatically resolves the FQNs and fixes last-mile syntax errors in 50 partial code samples collected from Stack Overflow with high success rates, without requiring any program analysis. The correct execution of the unit, module, and PCR-Chain demonstrates the effectiveness of the prompt design, prompt composition, and prompt architecture.**

Website:https://github.com/SE-qinghuang/PCR-Chain

Demo Video: https://youtu.be/6HGRNdc2_JE

***Index Terms*—In-context Learning, Pre-trained Language Model, Frozen Copilot, AI Chain, Hierarchical Prompts**

I. Introduction

Partial code from API documentation and Q&A site (e.g., Stack Overflow) is frequently reused in programming tasks [1]–[6]. This partial code cannot be compiled because it contains non-fully qualified names (non-FQNs) [7]–[11] and undeclared receiving objects (termed as cannot-be-resolved simple names), as well as last-mile syntax errors [12] (e.g., unbalanced parentheses, missing commas, missing quotes). Normally, developers manually repair partial code in two steps: FQN Inference (i.e., infer the missing FQNs) and Syntax Error Fix (i.e., check and fix last-mile syntax errors).

However, the two steps can be completed with partial program analysis techniques. For example, [13] makes type inferences based on abstract syntax tree, and [12] fixes syntax errors based on program synthesis. Recently, studies [14]–[17] on code naturalness shows code can be seen as text, and spawns a surge of pre-trained code models (PCMs) (e.g., CodeBERT [18], CodeT5 [19], Copilot [20]). By gradient-updating the weights of PCMs on a small set of instruction-like prompts, which are text strings with some unfilled slots, the task-agnostic PCMs adapt to specific tasks [18], [21]. This is known as supervised fine-tuning [22]–[25]. Once fine-tuend, the PCMs can fill in the missing information and produce the final output, as seen in studies like FQN-prompt-tuning applied to CodeBERT for FQN inference [7] and a fine-tuned programming language model fixing the syntax errors [26].

In contrast, there is a much more lightweight approach: in-context learning proposed by Brown et al. [27]. It uses behavior-describing prompts that describe the task and provide input-output examples, allowing giant PCMs with frozen weights to imitate the desired behavior and complete downstream tasks solely on the examples [28]–[31]. For example, in a question-answering task, the prompt might state, "The input is a question and the output is an answer to the question." along with examples of questions and answers. When fed to the frozen PCM, it can answer new questions by mimicking the behavior characteristic described in the prompt, without any weight updates [32]–[34].

In this paper, we use in-context learning to solve the problem of Java partial code not being compiled directly when reused, as shown in Fig. 1. First, we divide Partial Code Repair into FQN Inference and Syntax Error Fix. FQN Inference is further subdivided into SimpleName Extraction, SimpleName to FQN, and FQN Supplement, while Syntax Error Fix is subdivided into Code Check and Code Fix. This is a hierarchical task breakdown where the bottom tasks are

†Qing Huang and Jiahui Zhu are the first co-authors.
*Changjing Wang is corresponding author.

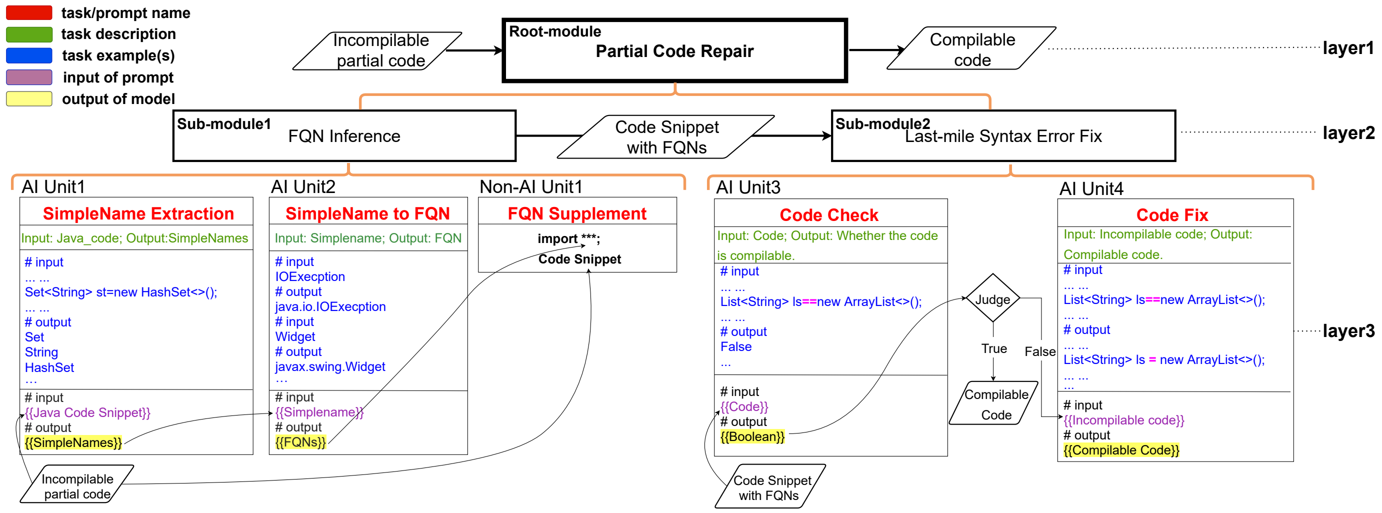

Fig. 1. Prompt Architecture and Design for PCR-Chain.

units and the non-bottom tasks are modules. Furthermore, the sequential structure is reflected in the execution of SimpleName Extraction, SimpleName to FQN, and FQN Supplement; the conditional structure is reflected in the execution of Code Fix, which is conditioned on the result of Code Check. Second, driven by in-context learning, we create a prompt (as shown in Fig. 1) on frozen Copilot (a frozen giant PCM) for each task at the bottom. This prompt depicts the current task's behavior characteristics, which Copilot then automatically imitates to finish the task. That is, each unit that performs the bottom tasks is prompt-based AI-driven.

Inspired by the above practice, at the global level, we propose a prompt architecture that fuses three design ideas: hierarchical task breakdown (i.e., divide a module into submodules or units from top to bottom), prompt composition (i.e., connect modules or units in a sequential or conditional structure), and a mix of prompt-based AI and Non-AI units (i.e., design executable AI or non-AI units). At the local level, we also present a prompt design (i.e., develop a task-appropriate prompt). Based on the prompt architecture and design, we develop a partial code reuse tool called PCR-Chain.

We use 50 uncompilable partial codes from SO to evaluate the accuracy of each unit, each model, and our PCR-Chain. Each unit's accuracy is above 0.82 or up to 0.97 (i.e., SimpleName Extraction unit), and each module's accuracy is above 0.85. This indicates that either prompt design or prompt composition design is effective. The PCR-Chain's accuracy is 0.70, indicating that the prompt architecture effectively resolves FQNs and fixes last-mile syntax errors.

This paper makes the following contributions:

- To the best of our knowledge, we are the first to propose the prompt architecture, which combines three global design ideas: hierarchical task breakdown, prompt composition, and a mix of prompt-based AI and non-AI units, rather than a simple AI chain.
- We present a prompt design at the local level for generating a task-appropriate prompt.
- We stand on the frozen giant PCM's shoulder and use a lightweight in-context learning method to resolve FQNs and fix syntax errors in partial code. When compared to the supervised fine-tuning approach, our method eliminates the need for special model tuning and deployment.
- The successful completion of the unit, module, and PCR-Chain demonstrates the efficacy of the prompt design, composition, and architecture, to resolve FQNs and fix last-mile syntax errors.

II. APPROACH

We design an underlying approach to assist the implementation of PCR-Chain, which involves a prompt architecture and a prompt design.

A. Prompt Architecture

The prompt architecture is framed by three design ideas: hierarchical task breakdown, prompt composition, and a mix of prompt-based AI and non-AI units.

1) Hierarchical Task Breakdown: This design is based on function analysis; according to different functions or different performances of the same function in a given range, the task can be divided from top to bottom into more fine-grained subtasks, or even a series of executable atomic units. The previous layer task is refined in the next layer task. Based on this design, PCR-Chain has the three-layer architecture.

As shown in Fig. 1-layer 1, the root module is Partial Code Repair that resolves FQNs of simple names and fix last-mile syntax errors in the partial code so that the incompilable partial code (i.e., input) becomes compilable code (i.e., output).

As shown in Fig. 1-layer 2, the root module is divided into two sub-modules because it performs two distinct functions successively. One sub-module is FQN Inference (i.e, resolve FQNs of undeclared receiving objects and non-FQNs in partial code); another is Syntax Error Fix (i.e, check and fix last-mile syntax errors [12] in the code).

As shown in Fig. 1-layer 3, the first sub-module is further divided into three units: SimpleName Extraction (i.e., extract the simple names of undeclared receiving objects and non-FQNs in the partial code), SimpleName to FQN (i.e., convert simple names to FQNs), and FQN Supplement (i.e., add import statements with FQNs at the start of the partial code). The second sub-module is further subdivided into two units: Code Check (i.e., determine whether or not the code contains syntax errors) and Code Fix (i.e., fix the syntax error).

2) Prompt Composition: This is a flexible design that connects units or modules into a new module with one or more of sequential or conditional structure. A sequential structure is a concatenation of modules or units, where the output of the previous module or unit is the input of the subsequent one. The conditional structure has a judgment condition and determines which subsequent module or unit to execute as a branch based on the outcome of the judgment.

Based on this design, three units in sub-module1 are connected in a sequential structure so that PCR-Chain first extracts simple names, then converts the simple names to FQNs, finally adds import statements with FQNs, whereas two units in sub-module2 are connected in a conditional structure so that PCR-Chain checks the code and fixes it if it contains syntax errors.

3) Mix of prompt-based AI and Non-AI Units: Two types of units are designed: one for AI and one for non-AI. The AI unit is appropriate for some tasks with uncertain execution logic or fuzzy matching. For example, we can only list some heuristic rules for extracting entities or entity relations from ever-changing natural language. Benefiting from the development of PCMs, syntactic or semantic knowledge packed in the PCMs can be transferred to the downstream SE tasks with uncertainty and fuzziness [22]. We implement a prompt-based AI unit using in-context learning, that is, we employ a prompt to a giant PCM (e.g., Copilot [20]) to specify the behavior characteristics of a downstream task that includes a task description and/or a few task demonstrations, and then asks the PCM to complete further instances of the task by mimicking the behavior characteristics. To quickly configure four prompt-based AI units (see Fig. 1), we devise a three-step pipeline:

- A prompt is a description and examples that represent the behavior characteristics of an AI unit task.
- A frozen Copilot is configured for the configured prompt by setting the frozen Copilot's parameters, e.g., temperature and maximum token as well as the current prompt.
- The configured frozen Copilot is asked to work by imitating the behavior characteristics.

The non-AI unit is appropriate for some tasks with certain execution logic or precise matching, such as pre-processing (case conversion, camel case naming) and post-processing (output merging, output verification). We implement a non-AI unit using a regular program function. FQN Supplement in Fig. 1 is an example, as it adds the output of the previous AI unit (*import statements with FQNs*) at the start of the code.

B. Prompt Design

The prompt is important for in-context learning because a giant PCM (e.g., Copilot [20]) works by learning the behavior characteristics depicted in a prompt. Prompt design involves prompt interaction design and prompt content design.

1) Prompt Interaction Design: As shown in Fig. 1-layer 3, four prompts are linked in a sequential, conditional structure to create a human-AI interaction, which is guided by continuous interaction with the PCM to solve problems like a human.

2) Prompt Content Design: In Fig. 1-layer 3, four prompts are specialized for four AI units, with a free-form format that includes task descriptions and input/output examples.

- SimpleName Extraction: This prompt depicts the behavioral characteristics of extracting the simple names of the undeclared receiving objects and non-FQNs in the partial code. It describes the task as "The input is java code, the output is simplename in java code". It also provides examples of as many forms of simple names as possible, such as "List", "List<>", "List[]", and "List()". For example, "*Set, String, HashSet*" are extracted from the code snippet "*... Set<String> st = new HashSet<>();...*".
- SimpleName to FQN: This prompt depicts the behavioral characteristics of converting simple names to FQNs. It describe the task as "The input are simplenames, the output are FQNs.". It also provides examples of converting various forms of simplename list to FQN. For example, "*IOException*" is converted to "*java.io.IOException*".
- Code Check: This prompt depicts the behavioral characteristics of checking whether or not the code has syntax error. It describes the task as "The input is code, the output is whether the code compilable.". It also includes examples of code with and without syntax errors. For example, the code snippet "*... List<String> ls == new ArrayList<>();...*" has syntax error.
- Code Fix: This prompt depicts the behavioral characteristics of fixing the syntax error. It describes the task as "The input is incompilable code , the output is compilable code." It also provides examples of illustrating code with a syntax error and its corrected version. For example, the uncompilable code "*... List<String> ls == new ArrayList<>();...*" is converted to the compilable code "*... List<String> ls = new ArrayList<>();...*".

III. Experiment

In this section, we evaluate the accuracy of each unit, each module, and the overall PCR-Chain framework.

A. Dataset & Metric

In this experiment, we use the Short-SO dataset from Stack Overflow by Huang et al. [7], containing 120 uncompilable partial code snippets from which we randomly selected 50 snippets. The evaluation metric for PCR-chain is accuracy.

B. Result and Analysis

1) Units Accuracy: The accuracy for each unit are shown in the second column of Table I. For **AI Unit of FQN**

TABLE I
EVALUATION RESULTS OF MULTIPLE SCENARIOS

AI Units	Acc	Modules	Acc	Architecture	Acc
SimpleName Extraction	0.97	FQN Inference	0.85	PCR-Chain	0.70
SimepleName to FQN	0.92				
Code Check	0.82	Syntax Error Fix	0.92		
Code Fix	0.84				

inference Module, the unit *SimpleName Extraction* correctly predicts 208 simple names out of 215 contained in the 50 code snippets, and the accuracy is 0.97. The unit *SimpleName to FQN* correctly inferred 190 FQNs for 215 simple names, and the accuracy is 0.88. For **AI Units of Syntax Error Fix Module** evaluation, Of the 50 code snippets, 25 have syntax errors and 25 do not. The unit *Code Check* correctly predicts 22 code snippets with syntax errors and 19 code snippets without syntax errors, and the accuracy is 0.82. For the 25 code snippets with syntax errors, the unit *Code Fix* fixes 21 code snippets correctly, and the accuracy is 0.84.

2) Modules Accuracy: The accuracy for each module is shown in the fourth columns of Table I. For **FQN Inference Module**, among 215 simple names, 183 simple names were correctly found and inferred the corresponding FQN with an accuracy of 0.85. For **Syntax Error Fix Module**, it accepts 50 code snippets and returns 46 code snippets with no syntax errors, with an accuracy of 0.92. Note that the accuracy of this module is greater than its two sub-units. The reasons are as follows: The unit Code Check made a mistake in identifying 6 code snippets without syntax errors as those with errors, leading to a decrease in its accuracy. On the other hand, the unit Code Fix successfully corrected the code snippets that were identified as having errors, resulting in an improvement in the module's accuracy.

3) PCR-Chain Accuracy: The accuracy for *PCR-Chain* is shown in the sixth columns of Table I. PCR-Chain fixes 35 out of 50 uncompilable partial codes to compilable, with an accuracy of 0.70.

Our evaluation assesses each unit, module, and the overall PCR-Chain. The high accuracy of the units confirms the usefulness of the prompt design and lays the foundation for high-quality modules. The successful execution of modules shows the value of prompt composition in linking units to achieve superior outcomes for higher-level tasks. The efficient functioning of PCR-Chain validates the utility of the prompt architecture in breaking down tasks into hierarchical units and modules, connecting them, and blending AI and non-AI unints. PCR-Chain also effectively resolves FQNs and fixes last-mile syntax errors.

IV. RELATED WORK

Partial code is prevalent in online resource (e.g., Stack Overflow), and developers frequently copy-paste it into Integrated Development Environments (IDEs) for reuse. However, the unresolved type and last-mile syntax errors prevent partial code from compiling. In the past, partial code program analysis was used to solve this issue, but it was limited by a high compilation overhead. [13] makes type inferences based on abstract syntax tree, and [12] fixes syntax errors based on program synthesis. Built on source code naturalness, recent approaches have overcome this by fine-tuning a large language model as a neural knowledge base of code elements using the "pre-train, prompt and predict" paradigm from raw source code. This minimizes the hte need for code compilation and eliminates the limitations of partial code program analysis. For example, Huang et al. utilize the prompt-tuned PCM to resolve types [7], and Jiang et al. use a PCM fine-tuned by an automated program repair task to fix syntax errors [26].

Our approach stands on the shoulder of the frozen PCM, which differs from the supervised fine-tuning described above in three ways: (1) Data Volumn. Jiang et al. [26] use 2.72 million examples while we only need 6 examples. Huang et al. [7] utilize 361 thousand prompts, each with one example, while we utilize two prompts with three examples each. (2) Chain of Thoughts (CoT). Unlike the traditional approach of presenting type inference as a one-step, fill-in-the-blank task, we adopt a more nuanced approach by dividing type inference into smaller and more manageable steps through the use of CoT. Each step operates independently and the outputs serve as inputs for the next, resulting in a cohesive Al chain. (3) Context Sensitivity. Without Al chain, Huang's method is context-sensitive, requiring FQNs in the surrounding context of to-be-infer simplename for best results. On the other hand, with the Al chain, we extract the simple names from the partial code and infer the FQNs independently, making type inference unrelated to context.

In-context learning, like our approach, is used for other purposes. Wang et al. [35] employ an informative prompt for a single-step task, and [36]–[38] chain prompts to break a multiple-task down into sub-tasks. PCR-Chain goes deeper than a prompt chain, involving a prompt architecture (hierarchical task breakdown, prompt composition, and a mix of AI and non-AI units) and a prompt design.

V. CONCLUSION AND FUTURE WORK

In this paper, we are first to propose the global-level prompt architecture and the local-level prompt design, rather than a simple AI chain. Supported by them, we implement PCR-Chain, an in-context learning-based partial code reuse tool with four prompts on frozen Copilot. Our experiments demonstrate the effectiveness of prompt architecture, prompt design, and PCR-Chain. The tool's source code and dataset are publicly available on GitHub, allowing other researchers to replicate and extend our work. The video will demonstrate how to use our tool. Although PCR-Chain is limited to the Java programming language, it is possible to adapt its use to other programming languages because in-context learning requires only a small number of examples in the prompt.

In the future, we intend to expand PCR-Chain into a systematic and configurable AI module framework, allowing us to build and organize a number of AI or non-AI modules to perform various tasks such as single-step tasks, multi-step tasks and nested-step tasks.

REFERENCES

[1] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522, 2010. I

[2] Margaret-Anne Storey, Leif Singer, Brendan Cleary, Fernando Figueira Filho, and Alexey Zagalsky. The (r) evolution of social media in software engineering. *Future of software engineering proceedings*, pages 100–116, 2014. I

[3] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, 126:57–84, 2017. I

[4] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th working conference on mining software repositories*, pages 102–111, 2014. I

[5] Le An, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. Stack overflow: A code laundering platform? In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 283–293. IEEE, 2017. I

[6] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. Stack overflow in github: any snippets there? In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 280–290. IEEE, 2017. I

[7] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code. *arXiv preprint arXiv:2208.05361*, 2022. I, III-A, IV

[8] Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. Semantic patches for java program transformation (experience report). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. I

[9] CM Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K Roy. Learning from examples to find fully qualified names of api elements in code snippets. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 243–254. IEEE, 2019. I

[10] Qing Huang, Dianshu Liao, Zhenchang Xing, Zhiqiang Yuan, Qinghua Lu, Xiwei Xu, and Jiaxing Lu. Se factual knowledge in frozen giant code model: A study on fqn and its retrieval. *arXiv preprint arXiv:2212.08221*, 2022. I

[11] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Zhengkang Zuo, Changjing Wang, and Xin Xia. 1+1¿2: Programming know-what and know-how knowledge fusion, semantic enrichment and coherent application. *ArXiv*, abs/2207.05560, 2022. I

[12] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013. I, II-A1, IV

[13] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 313–328, 2008. I, IV

[14] Premkumar T. Devanbu. On the naturalness of software. *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847, 2012. I

[15] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51:1 – 37, 2018. I

[16] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the" naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439, 2016. I

[17] Ahmed Khanfir, Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. Codebert-nt: code naturalness via codebert. *arXiv preprint arXiv:2208.06042*, 2022. I

[18] 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. *ArXiv*, abs/2002.08155, 2020. I

[19] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021. I

[20] "github copilot. your ai pair programmer" [online]. available:https://copilot.github.com/. I, II-A3, II-B

[21] Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, pages 10799–10808. PMLR, 2020. I

[22] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hairong Jin. What do they capture? - a structural analysis of pre-trained language models for source code. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2377–2388, 2022. I, II-A3

[23] Guangyuan Shi, Jiaxin Chen, Wenlong Zhang, Li-Ming Zhan, and Xiao-Ming Wu. Overcoming catastrophic forgetting in incremental few-shot learning by finding flat minima. *Advances in Neural Information Processing Systems*, 34:6747–6761, 2021. I

[24] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018. I

[25] Ananya Kumar, Aditi Raghunathan, Robbie Jones, Tengyu Ma, and Percy Liang. Fine-tuning can distort pretrained features and underperform out-of-distribution. *arXiv preprint arXiv:2202.10054*, 2022. I

[26] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021. I, IV

[27] Tom Brown, Benjamin Mann, Amanda Ryder, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. I

[28] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021. I

[29] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. Metaicl: Learning to learn in context. *arXiv preprint arXiv:2110.15943*, 2021. I

[30] Stephanie CY Chan, Adam Santoro, Andrew K Lampinen, Jane X Wang, Aaditya Singh, Pierre H Richemond, Jay McClelland, and Felix Hill. Data distributional properties drive emergent in-context learning in transformers. *arXiv preprint arXiv:2205.05055*, 2022. I

[31] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022. I

[32] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020. I

[33] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. I

[34] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher 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. *ArXiv*, abs/2005.14165, 2020. I

[35] Yizhong Wang, David Mishra, et al. Benchmarking generalization via in-context instructions on 1,600+ language tasks. *arXiv preprint arXiv:2204.07705*, 2022. IV

[36] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *CHI Conference on Human Factors in Computing Systems*, pages 1–22, 2022. IV

[37] Simran Arora, Avanika Narayan, Mayee F Chen, Laurel J Orr, Neel Guha, Kush Bhatia, Ines Chami, Frederic Sala, and Christopher Ré. Ask me anything: A simple strategy for prompting language models. *arXiv preprint arXiv:2210.02441*, 2022. IV

[38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022. IV