RustGen: An Augmentation Approach for Generating Compilable Rust Code with Large Language Models

Xingbo Wu [*1] **Nathanaël Cheriere** [*1] **Cheng Zhang** [1] **Dushyanth Narayanan** [1]

Abstract

Foundation models show an impressive ability to write code snippets. However, there are still challenges when generating code for resource-poor programming languages. In this work, using Rust as an example, we tackle these challenges through in-context learning, with additional components that feed back compile errors to the LLM until it converges on a runnable code that is free of several common programming errors. We describe the specific techniques that allow us to do this – history-based search, prompt engineering, and syntax-based skeletonization – and evaluate their benefits on a set of code generation tasks in Rust.

1. Introduction

Large Language Models (Zhao et al., 2023; Brown et al., 2020; OpenAI, 2023; Bubeck et al., 2023) are disrupting the world of software development through their ability to generate code, e.g. using services such as Copilot.[1] Typically these are used to generate code snippets that must then be reviewed, accepted, integrated, and tested by the human developer (Svyatkovskiy et al., 2019; Lin et al., 2017; Barke et al., 2023). Code generated by current LLMs often does not compile "out of the box" – for example, it can contain syntax errors, type errors, and missing dependencies. Fixing these is currently left to the human developer. Similarly writing unit tests, running the tests, and applying fixes when tests fail, all require significant human interaction.

Our vision is for AI models to generate systems code in a way that compiles out of the box, is testable, and scales to large code bases. We believe that repeated human interaction with the LLM, trying to "get it to do the right thing", dilutes the benefit of using LLMs for code generation – ideally this would be an automated task and not a manual one.

We would like to use LLMs unmodified as a black-box inference service, rather than training new models or fine-tuning existing large models (Ouyang et al., 2022; Le et al., 2022). This lets us trade training and fine-tuning costs for inference costs, which are orders of magnitude lower. Thus, we examine different in-context learning approaches to automate the process. by introducing additional components that enforce desirable properties on the code output by the LLM. E.g., if the code does not compile, we can automatically extract the compilation error and its context, and prompt the LLM to generate a fix. By doing this iteratively, the LLM can finally generate an output that has the desired properties.

We pick Rust as the target programming language as it is well-suited for high-performance and safety-critical large software systems. To date, most of the research on AI-generated code has focused on dynamically typed languages such as Python (Svyatkovskiy et al., 2019; Chen et al., 2021; Siddiq et al., 2022; Li et al., 2023a; Guo et al., 2022). In Python, type errors and many other errors are only detected at run time, making testing and debugging more complicated and ultimately reducing reliability. The Python garbage collector also introduces performance overheads that are avoided by C/C++ and Rust (Zhang et al., 2022; Ismail & Suh, 2018). Rust is a highly reliable language because it is strongly typed, has a ownership model for memory safety, and does not have a garbage collector (Balasubramanian et al., 2017; Bugden & Alahmar, 2022). Many types of errors can be detected by the Rust compiler and be fixed before the code is run. However, generating runnable Rust code is challenging for LLMs because the code must meet strict compile-time requirements (Zhu et al., 2022)and because training data for Rust is relatively scarce compared to Python or C/C++.

In this paper we describe RustGen, a prototype system that augments an LLM such as GPT-3.5 or GPT-4. Given an English-language description of the desired output code, RustGen repeatedly invokes the LLM until it satisfied the desired properties – e.g., error-free compilation – or a resource bound is reached. It tracks the entire history of

[*]Equal contribution [1]Microsoft Research, Cambridge, United Kingdom. Correspondence to: Xingbo Wu <xingbowu@microsoft.com>, Nathanaël Cheriere <nathanael.cheriere@microsoft.com>.

Workshop on Challenges in Deployable Generative AI at International Conference on Machine Learning (ICML), Honolulu, Hawaii, USA. 2023. Copyright 2023 by the author(s).

[1]https://github.com/features/copilot

interaction with the LLM, and uses this to generate new requests to the LLM to improve the code in this direction.

This paper makes the following contributions:

- We describe RustGen, an architecture and prototype for generating Rust modules from a natural language description.
- We describe the techniques that let us achieve this – history-based search of a candidate pool, and skeletonization to extract context.
- We show that in context learning utilizing compiler output significantly improves the success rate (from 2/10 to 8/10) of the LLM at generating runnable Rust source code.

2. Related work

AI-based models for program synthesis have been studied extensively in the literature (Svyatkovskiy et al., 2019; Chen et al., 2021; Li et al., 2022; Nijkamp et al., 2023), as well as approaches to automatic editing (Li et al., 2023a; Fried et al., 2023; Zhang et al., 2023) and bug identification (Allamanis et al., 2021; Pradel & Sen, 2018).

There is also extensive work on LLMs' code generation and repair capability through training (Lachaux et al., 2021; Chen et al., 2021; Li et al., 2022; Ciniselli et al., 2021; Drain et al., 2021; Li et al., 2023b), fine-tuning (Peng et al., 2023; Wei et al., 2022; Gao et al., 2021; Ouyang et al., 2022; Le et al., 2022), as well as guides and patterns for human prompting of LLMs for better code generation (DAIR.AI, 2023; Liu & Chilton, 2022; White et al., 2023; Reynolds & McDonell, 2021). In contrast we focus on the specific problem of automated repair using in-context learning using an unmodified, state-of-the-art LLM.

Most previous work on code repair has focused on high-resource languages such as Python, Java, and sometimes C/C++ (Jin et al., 2023; Gupta et al., 2017; Prenner & Robbes, 2021; Chen et al., 2023). Recent studies have also explored fully automated code synthesis including code repair for these high-resource languages, with LLMs using mix of sampling, ranking, and searching (Liventsev et al., 2023; Olausson et al., 2023).

We focus on Rust which differs in several key aspects. It is resource-poor with respect to data sets for training and fine-tuning. This also means that unit tests for generating feedback are not widely available; in addition, the challenge in generating Rust code is that of compile-time errors which are often difficult to understand and fix. Thus instead of using example-based or testing-based feedback for in-context learning, we focus on compiler errors as a source of feedback.

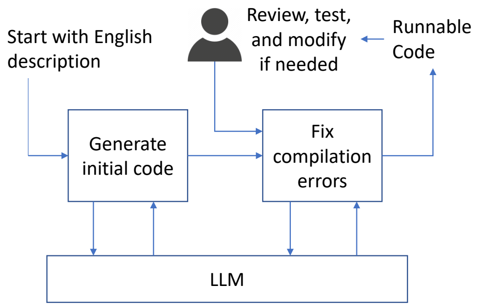
Start with English description
Review, test, and modify if needed
Runnable Code
Generate initial code
Fix compilation errors
LLM

Figure 1. RustGen code generation workflow

3. Design

Broadly, RustGen allows a developer to turn a high-level English description of a code module into runnable code for that module. Fig. 1 shows the high-level workflow for a developer writing a new piece of code using RustGen. After the initial code generation from an English description, the system repeatedly invokes the LLM and then the compiler until the code compiles. Finally, the visible output is the runnable code for testing. The initial code generation can optionally include tests, which might produce more compile errors, but it can make testing easier than writing tests from scratch. In this paper we focus on making code compiled.

The following introduces the algorithm of the error-fixing loop that makes the code compiled (§3.1), composing prompts for generating and fixing code (§3.2), and two optional features for improving RustGen's success rate (§3.3).

3.1. The Error-fixing Loop

Starting from an English description, RustGen generates an initial candidate program by prompting the LLM. In some simple cases this code is already compile error free, and we proceed to the next stage. Otherwise we repeatedly iterate until we find a compile error-free candidate or exceed an iteration bound. A pseudo code of the error-fixing process is shown in Algo. 1. The prompt template used for error fixing is discussed in §3.2, with examples in the appendix.

RustGen maintains a current candidate program and its compile errors that should be fixed (*code* and *errors* in Algo. 1). At each iteration, it picks one error from the errors, which is the first one by default, and provides the code and the error message in an error-fixing template as a prompt to the LLM to return a code with the error fixed. A new candidate program is extracted from the LLM's output and compiled. Ideally, the LLM fixes the error and reduces the number of errors by one. If the new candidate program does not produce any compile errors, the loop will terminate successfully. Otherwise, the new candidate program will replace the current one, and the next iteration will try to fix it.

Algorithm 1 RustGen's Error-fixing Loop

```
Input: initial code $code$, iteration budget $budget$, error-fixing template $template$.
$pool = \{\}$
for _ in $1$ to $budget$ do
  if $code$ not in $pool$ then
    $errors = Compile(code)$
    if $len(errors) == 0$ then
      return $code$
    $t = 0$                                  ▷Temperature of the request
    $i = 0$                                  ▷Zero-based index of the error to fix
  else
                                             ▷We have seen the code before.
                                             ▷Fixing errors[i] will likely fail again.
    $errors, i, t = pool[code]$
    $i += 1$                                 ▷Try to fix a different error
    if $i > len(errors)$ then
      $i = 0$
      increase $t$
      if $t > temperature_{max}$ then
                                             ▷Every attempt failed, restart from a previous step.
        $code = RandomPick(pool)$
        Continue
  Set $pool[code] = errors, i, t$
  $error = errors[i]$
                                             ▷See §3.3 for the optional skeletonization, default=off
  if Skeletonization then
    $code, bodies, error' = Skel(code, error)$
    $error = error'$                         ▷Use the updated error message.
  $code = AskLLM(template, code, error, t)$
  if Skeletonization then
    $code = DeSkel(code, bodies)$
  ▷If multiple answers are requested, deskeletonize all answers
  if necessary, pick one as code, and add the rest to the pool.
return Failed
```

When being asked for a fix, the LLM may return the same code. This can cause RustGen to get stuck in a loop, repeatedly failing to fix the same candidate. Similarly, when the LLM suggests a fix that has already been seen, the fixing loop can also struggle to make progress. To address these issues, RustGen maintains a candidate pool to record all the candidates that have been seen. The pool is indexed by code contents and stores each candidate's compile errors and information about which errors have been tried to fix. This lets us detect a second attempt with the same candidate and error.

To avoid infinite loops, we implement three mechanisms to improve the search when a second attempt is detected. Firstly, we retry the current candidate but attempt to fix a different error. If we have looped through all errors for a given candidate then we increase the LLM's *temperature* for that candidate. Temperature is an LLM API parameter which controls the level of non-determinism of the output. The initial temperature of a candidate is always set to zero, which in our experience gives a deterministic LLM inference – identical inputs give identical outputs. Secondly, if a candidate fails to compile even after trying it at the highest temperature, a different candidate is selected at random from the pool to continue the fixing. Finally, We configure an LLM API parameter to request multiple fixes at a time. When multiple fixes are returned, we pick one as the new candidate for fixing and add the other candidates to the pool for later use. The temperature schedule, the number of attempts at each temperature, and the number of fixes per attempt, are configurable parameters.

```
You are an expert Rust programmer.
Implement a Rust structure for LRU
caching. Given a key, it should be able
to look up quickly whether the key is in
the cache. If not, it should install in
the cache. If the cache size exceeds a
limit, the least recently touched item
should be evicted.
In your response, use RUST_BEGIN and
RUST_END to delimit the Rust source code.
```

Figure 2. Example of prompt used to generate code

3.2. Prompt Engineering

Prompt engineering is often required to design and refine prompts for LLMs to generate accurate and relevant responses (White et al., 2023; Liu & Chilton, 2022; Reynolds & McDonell, 2021; Gao et al., 2021). Prompts usually require four elements (DAIR.AI, 2023): instruction, context, input data, and output indicator. In Fig. 2 which is a prompt used to do the code generation from an English written description, contains these 4 elements. The first line is the context, the instruction and input data are in bold text, and the last part is the instruction regarding the output format. The output format is particularly important because the LLM's response needs to be processed programmatically. The appendix contains another example of an error-fixing prompt and its LLM response.

These prompts are based on templates (text in thin font), that can be programmatically filled to generate a code according to an English description or to fix a compile error. In our experience, having a single main instruction leads to better results. For example, generating the code and comments at the same time creates codes that are harder to fix compared to generating the code only.

An LLM has a constraint, or a token budget, on how many tokens that can be used to represent a prompt and its answer. For example, GPT-3.5 has a limit of 4 K tokens. To estimate the token budget of a request we run the tokenizer locally on the prompt (with source code) and estimate the response size to be the same as the source code since we request for the fixed source code to be returned. In our experience, a line of code uses about 10 tokens on average, which means that fixing compile errors is limited to codes with less than 200 lines with GPT-3.5. GPT-4's token budget is significantly higher at 32 K, potentially enabling the generation and correction of larger codes.

3.3. Optional Features

RustGen also supports two pluggable features for improving the code fixing quality. The first feature is automatically generating comments to the original source code. The source code generated by the original prompt often does not contain any commments. This feature uses a separate prompt to ask the LLM to add comments to the code in hope to improve the quality of the context for fixing.

The second feature is skeletonization, which is to reduce prompt sizes by removing code that is irrelevant to the error in question, e.g., bodies of other functions than the ones with the error. On the one hand, it reduces the cost of using the LLM. On the other hand, it enables larger codes to be generated and fixed. The skeletonization approach retains all context that might be required for a given code snippet to compile correctly. This is a more conservative approach than more general context extraction approaches such as eWASH (Clement et al., 2021).

The skeletonization is implemented based on static syntax parsing. RustGen invokes the Rust parser (using the syn crate) to produce a syntax tree, from which it extracts the function signatures, their bodies, and the line numbers they cover. From this and the compile error messages RustGen identifies the functions involved in each error. All other functions have their body replaced by the macro `unimplemented!()` (an example can be found in the appendix). The replaced function bodies are stored and indexed by function name. This ensures that the unmodified functions generates the same compile errors as the original code, but with different line numbers. A skeletonized code returned by the LLM should be deskeletonized before it can be used locally. We run the Rust parser on the returned code to locate the functions that were changed to `unimplemented!()` and revert them to the original state. Algo. 1 shows where skeletonization is used.

4. Evaluation

We evaluate RustGen with a set of ten programming tasks and two LLM models, namely GPT-3.5 and GPT-4. This section aims to answer the following questions: Does the automated error-fixing loop improve the compilation success rate? How does each of the additional features help compared to the default configuration? What type of errors are easy or hard to be fixed?

We implement the baseline as asking the LLM to generate one sample of the source code without any error fixing. It is worth noting that asking the LLM to generate multiple samples of the initial source code may improve success rate for both the baseline and RustGen. However, sampling the initial source code is out of the scope of our evaluation on the effectiveness of the error-fixing loop.

The default RustGen implementation includes all the mechanisms described in §3.1. Additionally, we evaluate three RustGen configurations: (1) asking the LLM for ten fixes in each fixing interaction, (2) skeletonization, and (3) generating comments for the initial code. The ten programming tasks are Sorted array, Binary tree, Self-balancing binary tree, Red-black tree, Skip list, Prefix tree, Chaining hash table, Cuckoo hash table, Hopscotch hash table, and LRU cache.

The experimental results are shown in Fig. 3, with detailed number of interactions in the Appendix. Without asking for fixings (Baseline), GPT-3.5 produced one compilable source code, while GPT-4 produced only two. These results contrast with prior experiences with generating C and Python code (Liventsev et al., 2023), where the generated code is presumably already compilable. In our observation, the low baseline success rate for Rust is mainly due to the strict type and ownership requirements in Rust than C and Python. For example, common error types include missing trait annotations and breaking the immutability of variables. The Rust compiler often suggests correct fixes for missing trait annotations errors, and the LLM is often able to fix these errors based on the suggestions. However, many other errors, such as a type mismatch, are not easy to fix.

RustGen (default) was able to generate compilable source code for 6 tasks for both GPT-3.5 and GPT-4. Among the successful results, RustGen needs at least three LLM fixing interactions with GPT-3.5 when there were errors in the initially generated source code. For all but one task, GPT-4 fixed the errors with only one or two LLM interactions.

The two features, generating comments and skeletonization, have played a positive role on GPT-4. On the one hand, they help the LLM by either adding useful information about the context, or removing unrelated information from the context. On the other hand, GPT-4 was able to leverage the improved context quality to generate correct code. However, the same mechanisms negatively affected the results with GPT-3.5 with success rates reduced. We speculate that main reason is due to the limited capability, GPT-3.5 fails to take advantage of the extra information in comments and had difficulty dealing with the partial context format.

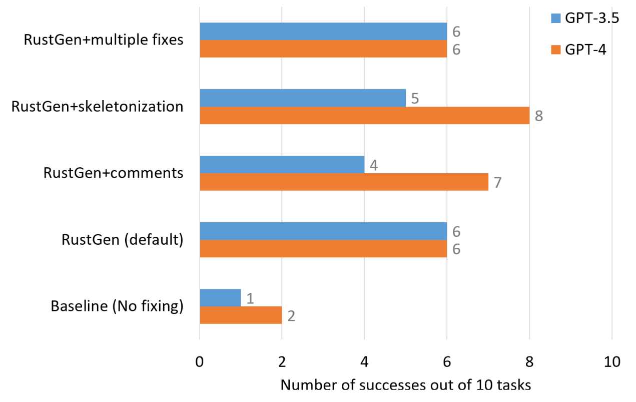

Figure 3. RustGen evaluation results

The configuration with asking for multiple (10) fixes tied with the default RustGen in terms of success rate. We have observed cases where asking for multiple fixes can lead to more or fewer LLM interactions. There is not a clear evidence on if this feature can improve the outcome, as the default RustGen is already carefully exploring the fixing space by avoiding loops and adjusting the LLM parameters.

In summary, the best result is with GPT-4 by turning on the skeletonization, which achieved a success rate of 8/10, a significant improvement compared to the baseline result of 2/10. While testing is beyond the scope of this paper, we manually inspect the generated codes and observed a few logical issues. For example, the skip list has the correct structure, but it works as a plain linked list at an O(N) time complexity. Unfortunately, these logical issues cannot be revealed by the Rust compiler or even simple tests.

5. Discussion and Future Work

We are still far from automatically generating a large system such as an OS kernel. Our results show that the generated code can have logic bugs and missing functionality that can only be captured by human inspection or by human-generated tests (Hendrycks et al., 2021). An important future direction is to combine compile-error repair with test-based repair (Jin et al., 2023; Gupta et al., 2017; Prenner & Robbes, 2021; Chen et al., 2023). However automatic generation of correct tests in Rust is again a challenging problem due to resource poverty. We have found that using the LLM to generate tests poses a new challenge – buggy tests that match buggy code (and hence pass). An interesting question is whether ”asking the LLM to repair itself” (Olausson et al., 2023), can solve this problem for a resource-poor language.

Context extraction remains a challenge at each step – as we try to automate more complex coding tasks and generate larger code bases, this will become increasingly important. Syntax-based context extraction such as the skeletonization used in RustGen, or eWASH (Clement et al., 2021), cannot scale to large complex codes with complex relationships spread throughout the source code. It is likely that more powerful techniques based on whole-program analysis will be needed here.

References

Allamanis, M., Jackson-Flux, H., and Brockschmidt, M. Self-supervised bug detection and repair. In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 27865–27876. Curran Associates, Inc., 2021. https://proceedings.neurips.cc/paper_files/paper/2021/file/ea96efc03b9a050d895110db8c4af057-Paper.pdf.

Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamari, Z., and Ryzhyk, L. System programming in Rust: Beyond safety. *SIGOPS Operating Systems Review*, 51(1):94–99, Sep 2017. https://doi.org/10.1145/3139645.3139660.

Barke, S., James, M. B., and Polikarpova, N. Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111, 2023. https://dl.acm.org/doi/abs/10.1145/3586030.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020. https://arxiv.org/abs/2005.14165.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M. T., and Zhang, Y. Sparks of Artificial General Intelligence: Early experiments with GPT-4, 2023. https://arxiv.org/abs/2303.12712.

Bugden, W. and Alahmar, A. Rust: The programming language for safety and performance. 2022. https://arxiv.org/abs/2206.05503.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. 2021. https://arxiv.org/abs/2107.03374.

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug, 2023. https://arxiv.org/abs/2304.05128.

Ciniselli, M., Cooper, N., Pascarella, L., Poshyvanyk, D., Penta, M. D., and Bavota, G. An empirical study on the usage of BERT models for code completion, 2021. https://arxiv.org/abs/2103.07115.

Clement, C. B., Lu, S., Liu, X., Tufano, M., Drain, D., Duan, N., Sundaresan, N., and Svyatkovskiy, A. Long-range modeling of source code files with eWASH: Extended window access by syntax hierarchy, 2021. https://arxiv.org/abs/2109.08780.

DAIR.AI. Prompt engineering guide. https://www.promptingguide.ai/, 2023. Accessed: 2023-07-07.

Drain, D., Wu, C., Svyatkovskiy, A., and Sundaresan, N. Generating bug-fixes using pretrained transformers. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*. ACM, Jun 2021. https://dl.acm.org/doi/10.1145/3460945.3464951.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis, 2023. https://arxiv.org/abs/2204.05999.

Gao, T., Fisch, A., and Chen, D. Making pre-trained language models better few-shot learners, 2021. https://arxiv.org/abs/2012.15723.

Guo, D., Svyatkovskiy, A., Yin, J., Duan, N., Brockschmidt, M., and Allamanis, M. Learning to complete code with sketches, 2022. https://arxiv.org/abs/2106.10158.

Gupta, R., Pal, S., Kanade, A., and Shevade, S. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pp. 1345–1351. AAAI Press, 2017. https://dl.acm.org/doi/10.5555/3298239.3298436.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps, 2021. https://arxiv.org/abs/2105.09938.

Ismail, M. and Suh, G. E. Quantitative overhead analysis for Python. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 36–47. IEEE, 2018. https://ieeexplore.ieee.org/document/8573512.

Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N., and Svyatkovskiy, A. InferFix: End-to-end program repair with LLMs, 2023. https://arxiv.org/abs/2303.07263.

Lachaux, M.-A., Roziere, B., Szafraniec, M., and Lample, G. DOBF: A deobfuscation pre-training objective for programming languages. In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 14967–14979. Curran Associates, Inc., 2021. https://proceedings.neurips.cc/paper_files/paper/2021/file/7d6548bdc0082aacc950ed35e91fcccb-Paper.pdf.

Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. H. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning, 2022. https://arxiv.org/abs/2207.01780.

Li, J., Li, Y., Li, G., Jin, Z., Hao, Y., and Hu, X. SkCoder: A sketch-based approach for automatic code generation. 2023a. https://arxiv.org/abs/2302.06144.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M.-H., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., Murthy, R., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder: may the source be with you!, 2023b. https://arxiv.org/abs/2305.06161.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, dec 2022. doi: 10.1126/science.abq1158. https://www.science.org/doi/10.1126/science.abq1158.

Lin, X. V., Wang, C., Pang, D., Vu, K., and Ernst, M. D. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01*, 2017.

Liu, V. and Chilton, L. B. Design guidelines for prompt engineering text-to-image generative models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pp. 1–23, 2022. https://dl.acm.org/doi/abs/10.1145/3491102.3501825.

Liventsev, V., Grishina, A., Härmä, A., and Moonen, L. Fully autonomous programming with large language models. 2023. https://arxiv.org/abs/2304.10423.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. CodeGen: An open large language model for code with multi-turn program synthesis, 2023. https://arxiv.org/abs/2203.13474.

Olausson, T. X., Inala, J. P., Wang, C., Gao, J., and Solar-Lezama, A. Demystifying GPT self-repair for code generation, 2023. https://arxiv.org/abs/2306.09896.

OpenAI. GPT-4 technical report, 2023. https://arxiv.org/abs/2303.08774.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. Training language models to follow instructions with human feedback, 2022. https://arxiv.org/abs/2203.02155.

Peng, B., Li, C., He, P., Galley, M., and Gao, J. Instruction tuning with GPT-4, 2023. https://arxiv.org/abs/2304.03277.

Pradel, M. and Sen, K. DeepBugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. https://doi.org/10.1145/3276517.

Prenner, J. A. and Robbes, R. Automatic program repair with OpenAI's Codex: Evaluating QuixBugs. 2021. https://arxiv.org/abs/2111.03922.

Reynolds, L. and McDonell, K. Prompt programming for large language models: Beyond the few-shot paradigm, 2021. https://arxiv.org/abs/2102.07350.

Siddiq, M. L., Majumder, S. H., Mim, M. R., Jajodia, S., and Santos, J. C. An empirical study of code smells in transformer-based code generation techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 71–82. IEEE, 2022. https://ieeexplore.ieee.org/document/10006873.

Svyatkovskiy, A., Zhao, Y., Fu, S., and Sundaresan, N. Pythia: AI-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2727–2735, 2019. https://dl.acm.org/doi/10.1145/3292500.3330699.

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. Finetuned language models are zero-shot learners, 2022. https://arxiv.org/abs/2109.01652.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. A prompt pattern catalog to enhance prompt engineering with ChatGPT. 2023. https://arxiv.org/abs/2302.11382.

Zhang, K., Li, Z., Li, J., Li, G., and Jin, Z. Self-Edit: Fault-aware code editor for code generation, 2023. https://arxiv.org/abs/2305.04087.

Zhang, Q., Xu, L., Zhang, X., and Xu, B. Quantifying the interpretation overhead of python. *Science of Computer Programming*, 215:102759, 2022. https://dl.acm.org/doi/abs/10.1016/j.scico.2021.102759.

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y., and Wen, J.-R. A survey of large language models, 2023. https://arxiv.org/abs/2303.18223.

Zhu, S., Zhang, Z., Qin, B., Xiong, A., and Song, L. Learning and programming challenges of Rust: A mixed-methods study. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pp. 1269–1281, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3510003.3510164.

A. Prompt Examples.

Fig. 4 shows an example of error-fixing prompt. The corresponding answer from the LLM is shown in Fig. 5. An example of skeletonized code is shown in Fig. 6, where three functions have their body replaced with `unimplemented!();`.

```
CONTEXT:
You are an expert programmer. The language used is
Rust.

GOAL:
Return the entire code with the error corrected.

CODE:
...
impl<K, V> LruCache<K, V>
where
    K: Hash + Eq,
{
...
    // Attach the node to the head of the list
    self.attach(node);
    // Insert the node pointer into the hash map
    self.map.insert(node.key.clone(), node);
...
}
ERROR:
error[E0599]: no method named `clone` found for type
parameter `K` in the current scope
  --> src\\main.rs:86:38
   |
22 | impl<K, V> LruCache<K, V>
    |    - method `clone` not found for this type parameter
...
86 |     self.map.insert(node.key.clone(), node);
   |                              ^^^^^ method not found in `K`
   |
   = help: items from traits can only be used if the type
parameter is bounded by the trait
help: the following trait defines an item `clone`, perhaps
you need to restrict type parameter `K` with it:
   |
24 |     K: Hash + Eq + Clone,
   |                   +++++++
REQUEST:
Makes changes to the code to fix the error.
In your response, use "RUST_BEGIN" and "RUST_END" to
delimit the Rust source code.

SOLUTION:
```

Figure 4. Example of prompt used to fix compilation errors. The source code was shortened for presentation and the indentation of code and error was edited to fit in the paper.

```
RUST_BEGIN
...
impl<K, V> LruCache<K, V>
where
    K: Hash + Eq + Clone,
{
...
}
RUST_END
```

Figure 5. Response from the LLM to the request presented in figure 4. The source code was shortened for presentation.

```
struct SortedArray<T: Ord> {
    data: Vec<T>,
}
impl<T: Ord> SortedArray<T> {
    fn new() -> Self {
        Self { data: Vec::new() }
    }
    fn insert(&mut self, value: T) {
        unimplemented!();
    }
    fn lookup(&self, index: usize) -> Option<&T> {
        unimplemented!();
    }
    fn lower_bound(&self, key: &T) -> Option<&T> {
        unimplemented!();
    }
}
```

Figure 6. Example of skeletonized code. All function bodies except for `new()` have been replaced by the macro `unimplemented!();`.

B. Problem Descriptions

Descriptions of the ten problems are listed in the following ten figures.

```
Implement a simple sorted array data structure in Rust.
The data structure supports insert(), lookup(), and lower_bound() operations.
The lower_bound operation returns the smallest key greater than or equal to the given input key in the array, or
     returns None if no such key exists.
```

Figure 7. Problem 1: sorted array

```
Implement a simple binary tree data structure in Rust.
The binary tree supports insert, remove, lookup, and lower_bound operations.
The lower_bound operation returns the smallest key greater than or equal to the given input key in the tree, or
     returns None if no such key exists.
```

Figure 8. Problem 2: binary tree

```
Implement an self-balancing binary tree data structure in Rust.
The binary tree supports insert, remove, lookup, and lower_bound operations.
The lower_bound operation returns the smallest key greater than or equal to the given input key in the tree, or
     returns None if no such key exists.
```

Figure 9. Problem 3: self-balancing binary tree

```
Implement a red-black tree data structure in Rust.
In addition to the requirements imposed on a binary search tree the following must be satisfied by a red-black tree:
Every node is either red or black.
All NIL nodes are considered black.
A red node does not have a red child.
Every path from a given node to any of its descendant NIL nodes goes through the same number of black nodes.
The red-black tree supports insert, lookup, remove, lower_bound operations.
The lower_bound operation returns the smallest key greater than or equal to the given input key in the tree, or
     returns None if no such key exists.
```

Figure 10. Problem 4: red-black tree

```
Implement a skip list data structure in Rust.
The skip list supports insert, lookup, remove, lower_bound operations.
The lower_bound operation returns the smallest key greater than or equal to the given input key in the skip list, or
      returns None if no such key exists.
```

Figure 11. Problem 5: skip list

```
Implement a hash table in Rust from scratch. The hash table contains an array of hash buckets. Each hash bucket is a
      linked list. The hash table supports insert, remove, and lookup operations.
```

Figure 12. Problem 6: chaining hash table

```
Implement a Rust structure for LRU caching. Given a key, it should be able to look up quickly whether the key is in
     the cache. If not it should install in the cache. If the cache size exceeds a limit, the least recently touched
      item should be evicted.
```

Figure 13. Problem 7: LRU cache

```
Implement a prefix tree (trie) to store strings of arbitrary length. The strings are sorted in alphabetical order in
      the trie.
Each trie node contains a fixed-size vector to store child nodes and a boolean value to indicate if the node is the
     end of a key.
The trie should support the following operations: add, remove, contains, lower_bound, and prefix_match_count. The
     lower_bound operation returns the smallest key greater than or equal to the given input key in the tree, or
     returns None if no such key exists. The prefix_match_count method returns the count of keys in the trie that
     matches a prefix.
```

Figure 14. Problem 8: prefix tree

```
As an expert Rust programmer, follow these instructions to implement a Hopscotch hash table from scratch:
Create a HopscotchHashTable struct containing a usize field hop_range, a usize field table_size, and a Vec of size
     table_size+hop_range to store key-value pairs.
Initialize the Vec with Option types set to None for empty slots.

Implement the hash function using std::collections::hash_map::DefaultHasher, taking the key and table size as
     parameters. This function will apply the hash function and return the hashed index for the key.

Implement the insert_helper method, taking a key-value pair as parameter.
a. Calculate the hash index of the key.
b. Search for an empty slot in the range from hash_index to hash_index + hop_range.
c. If an empty slot is found, insert the key-value pair to the slot and return None.
d. randomly pick a slot in the search range and swap the key-value pair in the slot with the input key-value pair.
e. return the swapped-out key-value pair.

Implement the try_insert method, taking a key-value pair as parameter.
a. Initialize the current key-value pair to be the input key-value pair.
b. Call insert_helper with the current key-value pair.
c. If None is returned from insert_helper, return None.
d. Otherwise, update the current key-value pair with the one returned from insert_helper.
e. repeat from step b. if maximum number of retries has not been reached. Otherwise, return the current key-value
     pair.

Implement the insert_method, taking a key-value pair as parameter.
a. Initialize the current key-value pair to be the input key-value pair.
b. Search for the current key in the range from hash_index to hash_index + hop_range. If the key is found, return
     false.
c. Call try_insert with the current key-value pair.
c. If None is returned, return true. Otherwise, update the current key-value pair with the returned one, call the
     resize method, and repeat from step c.

Implement the resize method: Double the table_size. Put the current table aside and replace it with a new table of
     the new table_size+hop_range.
For every key in the original table, initialize the current key-value pair, call try_insert repeatedly with updating
      the current key-value pair, until None is returned.

Implement the lookup, remove, update methods.
```

Figure 15. Problem 9: hopscotch hash table

```
As an expert Rust programmer, follow these instructions to implement a cuckoo hash table with two hash functions and
      a single table structure:
Choose or implement two distinct hash functions that accept a seed value, such as using std::collections::hash_map::
     DefaultHasher with different seeds or implementing your own hash functions.
Create a CuckooHashTable struct with a single Vec to store key-value pairs and two seeds for the hash functions.
     Initialize the Vec with Option types set to None for empty slots.

Implement the hash method, taking the key, hash seed, and table size as parameters. This method will apply one of
     the hash functions based on the seed and return the hashed index for the key.

Implement the insert method:
the current key to be inserted is the input key.
a. Calculate the two positions for the current key using the two hash seeds.
b. Check if any of the positions contains the key. If the key exists, return FailureExists.
c. Check if any of the positions is empty. If an empty position is found, directly insert the new key-value pair
     into that position and return Success.
d. Call a helper method insert_internal to handle the insertion. If insert_internal returns Success, return Success.
e. Otherwise, a key-value pair to be inserted back to the table is returned by insert_internal. Call the resize
     method, and then repeat from step d. with the returned key-value pair as the input.

Implement the insert_internal method:
a. Calculate the two positions for the current key using the two hash seeds.
b. Check if any of the positions is empty. If an empty position is found, directly insert the new key-value pair
     into that position and return Success.
c. If both positions are occupied, pick one position using a random number, swap the current key-value pair with the
      existing key-value pair in the picked position.
d. Repeat the process from step a. with the new current key-value pair, until an empty slot is found. When a maximum
      number of iterations is reached, return FailureRetry with the current key-value pair.
The insert_internal method should also has a boolean retry_indefinitely parameter which force ignoring the maximum
     number of iterations check.

Implement the resize method for the CuckooHashTable struct: Put the current table aside and replace it with a new
     table twice as large. Call insert_internal with retry_indefinitely=true for all the key-value pairs to the new
     table. Then discard the old table.

Implement the search, update, and delete methods.
```

Figure 16. Problem 10: cuckoo hash table

C. Evaluation Results

The following table shows the detailed number of fixing interactions and final result of each task. Having zero fixing interactions means that the initially generated source code is already compilable. Termination with failure can be a result of reaching a dead end with an empty candidate pool or exceeding the maximum number of fixing interactions.

Table 1. Summary of Fixing interactions and final compilable results

LLM	Problem	RustGen default		+comments		+skeletonization		+multiple fixes	
		Interact.	Succ.	Interact.	Succ.	Interact.	Succ.	Interact.	Succ.
GPT-3	Binary tree	404	FALSE	404	FALSE	792	TRUE	303	FALSE
	Chaining hash table	8	TRUE	43	FALSE	18	FALSE	5	TRUE
	Cuckoo hash table	3	TRUE	156	FALSE	5	TRUE	3	TRUE
	Hopscotch hash table	16	TRUE	146	TRUE	22	TRUE	35	TRUE
	LRU cache	404	FALSE	404	FALSE	1111	FALSE	404	FALSE
	Red-black tree	6	TRUE	5	TRUE	1111	FALSE	6	TRUE
	Self-balancing binary tree	3	TRUE	8	TRUE	1111	FALSE	3	TRUE
	Skip list	404	FALSE	299	FALSE	1111	FALSE	303	FALSE
	Sorted array	0	TRUE	1	TRUE	0	TRUE	0	TRUE
	Prefix tree	5	FALSE	1	FALSE	4	TRUE	303	FALSE
GPT-4	Binary tree	38	TRUE	85	TRUE	131	TRUE	28	TRUE
	Chaining hash table	237	FALSE	75	TRUE	64	TRUE	65	FALSE
	Cuckoo hash table	1	TRUE	9	TRUE	1	TRUE	1	TRUE
	Hopscotch hash table	0	TRUE	1	TRUE	0	TRUE	0	TRUE
	LRU cache	1	TRUE	2	TRUE	1	TRUE	1	TRUE
	Red-black tree	99	FALSE	0	FALSE	27	TRUE	29	FALSE
	Self-balancing binary tree	22	FALSE	7	FALSE	53	TRUE	0	FALSE
	Skip list	65	FALSE	68	FALSE	257	FALSE	55	FALSE
	Sorted array	0	TRUE	1	TRUE	0	TRUE	0	TRUE
	Prefix tree	2	TRUE	22	TRUE	33	FALSE	2	TRUE