

# Fully Autonomous Programming with Large Language Models

Vadim Liventsev\*<sup>†</sup>  
v.liventsev@tue.nl  
TU Eindhoven &  
Philips Research  
The Netherlands

Anastasiia Grishina\*  
anastasiia@simula.no  
Simula & University of  
Oslo  
Norway

Aki Härmä  
aki.harma@philips.com  
Philips Research  
Eindhoven  
The Netherlands

Leon Moonen  
leon.moonen@computer.org  
Simula & BI Norwegian  
Business School  
Oslo, Norway

## ABSTRACT

Current approaches to program synthesis with Large Language Models (LLMs) exhibit a “near miss syndrome”: they tend to generate programs that semantically resemble the correct answer (as measured by text similarity metrics or human evaluation), but achieve a low or even zero accuracy as measured by unit tests due to small imperfections, such as the wrong input or output format. This calls for an approach known as Synthesize, Execute, Debug (SED), whereby a draft of the solution is generated first, followed by a program repair phase addressing the failed tests. To effectively apply this approach to instruction-driven LLMs, one needs to determine which prompts perform best as instructions for LLMs, as well as strike a balance between repairing unsuccessful programs and replacing them with newly generated ones. We explore these trade-offs empirically, comparing replace-focused, repair-focused, and hybrid debug strategies, as well as different template-based and model-based prompt-generation techniques. We use OpenAI Codex as the LLM and Program Synthesis Benchmark 2 as a database of problem descriptions and tests for evaluation. The resulting framework outperforms both conventional usage of Codex without the repair phase and traditional genetic programming approaches.

## CCS CONCEPTS

• **Software and its engineering** → *Software design engineering*; • **Computing methodologies** → *Neural networks; Model development and analysis; Search methodologies*.

## KEYWORDS

automatic programming, large language models, program repair

## ACM Reference Format:

Vadim Liventsev, Anastasiia Grishina, Aki Härmä, and Leon Moonen. 2023. Fully Autonomous Programming with Large Language Models. In *Genetic and Evolutionary Computation Conference (GECCO '23), July 15–19, 2023, Lisbon, Portugal*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3583131.3590481>

\* Both authors contributed equally to this research.

<sup>†</sup> Corresponding author.

## 1 INTRODUCTION

Automatic programming has been an important goal of the Artificial Intelligence field almost since its inception [1], promising to reduce the workload of software developers by automatically solving some of the tasks they face. More recently, program synthesis has emerged as an interpretable alternative [2] to black-box machine learning methods that lets human experts understand, validate and edit the algorithms generated by artificial intelligence. In addition to the scientific benefits of such knowledge, it extends the benefits of machine learning to domains, such as embedded systems where it is technically challenging [3] or healthcare where it is avoided for safety reasons [4, 5].

The predominant methodology in automatic programming has shifted from deductive programming [6, 7] to genetic and evolutionary methods [8] to, more recently, large autoregressive language models trained on corpora of source code due to their remarkable capability for zero-shot generalization [9]. However, even state-of-the-art models fine-tuned on a specific class of programming tasks still require a costly filtering step where the LLM outputs that do not compile or pass tests are discarded [10]. These outputs tend to be superficially similar to correct solutions [11] despite failing to produce the expected output, a phenomenon known as “near miss syndrome” or “last mile problem” [12].

Given these challenges, research in machine learning on source code [13] tends to focus on restricted domain-specific languages [14–16] or automating specific parts<sup>1</sup> of the software development process [19, 20] such as code search [21], code translation [22], detection of issues [23, 24], improvement [25] and repair [26] rather than fully autonomous programming in a programming language popular with human developers [27]. However, two recent innovations potentially make the latter task tractable.

One is *Synthesize, Execute, Debug* [28], a framework that attempts to bridge the “last mile” gap by introducing program repair into the program synthesis algorithm. A programming task is specified using both a natural language description and a set of input/output (I/O) pairs demonstrating what output is expected of the program, thereby combining text to code [29] and programming by example [30, 31] paradigms typical for competitive programming [32]. *Synthesize, Execute, Debug* creates a first draft program using a generative model, compiles and executes it with given input examples. This is followed by a program repair step to fix the identified errors.

Another relevant innovation is instruction-driven large language models [33]. Instruction-driven models use human feedback in their training process and admit two inputs: a source text (or code) and a textual command instructing the model to edit the source in a

<sup>1</sup> similarly to autonomous driving [17, 18]



This work is licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) license.

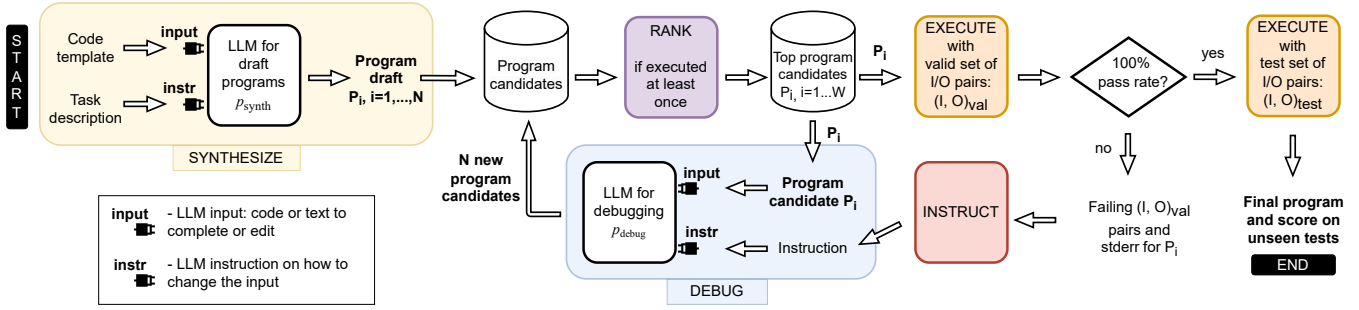


Figure 1: Framework for LLM-based Synthesize, Execute, Instruct, Debug, and Rank approach.

particular way, i.e., "summarize" or "translate to Python". These models have been shown to be highly successful in automatic program repair [34]. However, given the free-form nature of these instructions<sup>2</sup> how one should engineer instructions that maximize repair performance is an open question.

Section 2 presents a framework that adapts *Synthesize, Execute, Debug* to instruction-driven Large Language Models for solving programming tasks in an autonomous fashion. We discuss related work in Section 3, introduce experiments to establish optimal search and prompting strategies for this framework in Section 4. Finally, we demonstrate in Section 5 that our framework outperforms conventional automatic programming techniques, such as genetic programming and naive application of large language models that generate one solution per problem without updating it iteratively.

## 2 METHODOLOGY

The proposed framework, *Synthesize, Execute, Instruct, Debug and Rank*, or SEIDR,<sup>3</sup> is summarized in figure 1. To solve a programming task defined as a text description and a collection of I/O examples, we split I/O examples into prompt and validation sets and use the prompt set in a large language model to SYNTHESIZE a population of candidate solutions. We EXECUTE the solutions, test them against the validation set, generate a text description of the identified problems used to INSTRUCT a large language model to produce repaired candidate solutions similar to the way a human developer DEBUGs a program. We RANK the candidates by correctness measured by matching I/O pairs, discard the worst candidates, and repeat until a fully correct solution is found.

### 2.1 Ingredients

SEIDR makes use of 2 instruction-driven large language models for source code: a *synthesis* model  $p_{\text{synth}}$ (input, instr) and a *debugging* model  $p_{\text{debug}}$ (input, instr), as well as, optionally, a large natural language model  $p_{\text{text}}$ (input) that can be used for writing instructions for the code model. Each model is a highly parameterised probability distribution over the space of (input, instruction)-tuples with parameters estimated on a large diverse (i.e., non-task-specific) corpus. This stochastic nature of language models is an important

prerequisite for SEIDR, since it lets us sample batches of diverse candidate solutions from  $p_{\text{synth}}$ (input, instr),  $p_{\text{debug}}$ (input, instr), and  $p_{\text{text}}$ (input). We have chosen the state-of-the-art transformer models [36] for  $p_{\text{synth}}$ (input, instr),  $p_{\text{debug}}$ (input, instr), and  $p_{\text{text}}$ (input) in our experiments as described in Section 4.5. In general, SEIDR requires a sequence-to-sequence generative model for these blocks.

### 2.2 Synthesize

The framework starts with the SYNTHESIZE block, which is responsible for generating initial draft solutions to programming tasks to be repaired in the later stages of SEIDR. We start with a basic template for a chosen programming language that contains a number of standard library imports and an empty *main* function or this language’s equivalent thereof, see figure 2. We populate this template with a comment indicating a text description of a task at hand and several I/O examples from the prompt training set. We design the templates with guidelines by the authors of the language model [37] and prior work [38] in mind. We then sample  $N$  programs from  $p_{\text{synth}}$ (input, instr), setting input to the populated template and instruction to the problem description. We use temperature sampling with a monotonically increasing temperature schedule where  $i$ -th program is sampled with temperature  $t_i \approx \frac{i}{N}$  (approximate equality enables efficient implementation by means of batching). Thus, the sampling procedure for the first programs

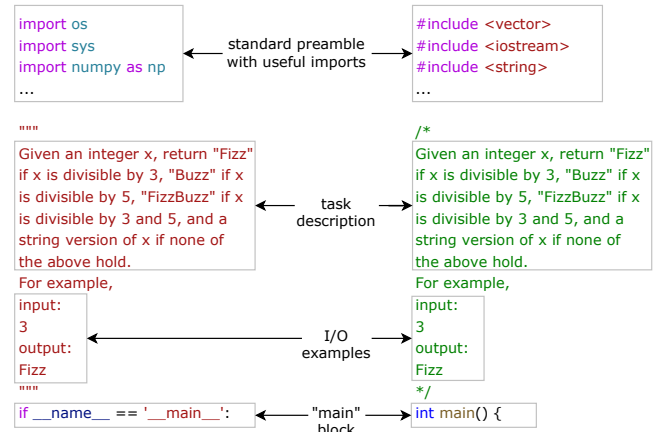


Figure 2: Anatomy of SYNTHESIZE templates

<sup>2</sup> Throughout this paper we avoid other definitions of *instruction*, such as *an individual operation in code*, to prevent ambiguity.

<sup>3</sup> seidr also refers to a type of Norse magic [35] pertaining to predicting and controlling the future, which we deem thematically appropriate.

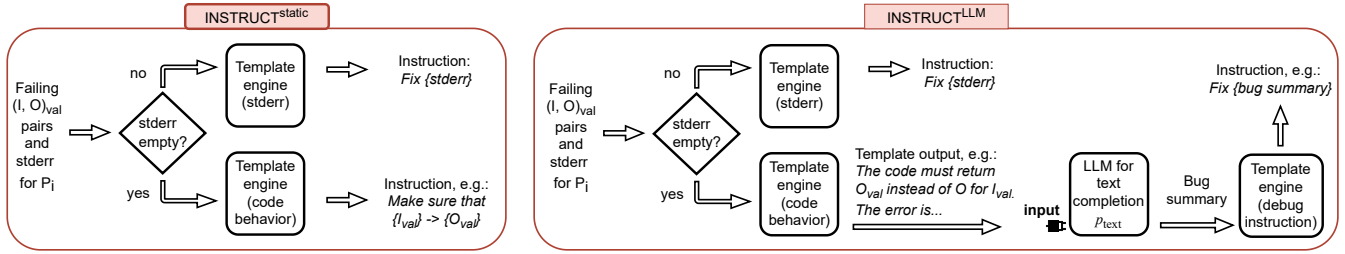


Figure 3: INSTRUCT blocks with and without an LLM.

approximates deterministic maximum likelihood estimation. Ultimately, this approach ensures that samples are diverse, but always contain the likeliest programs.

### 2.3 Execute

In the EXECUTE block, the programs are compiled (if necessary) and launched using the standard tools for the programming language. The program is run once for every I/O pair in the validation set. Its `stdin` stream receives all the input lines in a given input pair, and its `stdout` and `stderr` streams are captured and saved. We then measure the *score* of the program defined as accuracy over output lines, with  $O$  being the expected output, and  $n = \max\{|O|, |\text{stdout}|\}$ :

$$\text{score}(O, \text{stdout}) = \frac{\sum_i^n \mathbb{I}[\text{stdout}_i = O_i]}{n}$$

unless `stderr` is non-empty during compilation or execution, which is considered to indicate failure and is assigned a score of 0.

### 2.4 Instruct

The goal of the INSTRUCT block is to provide an instruction that summarizes a bug in the program candidate for  $p_{\text{debug}}$  (input, instr). The resulting instruction with the bug summary should indicate what requirement is violated and instruct the LLM to edit the candidate program so that the candidate meets the violated requirements. In SEIDR, we generate instructions using template engines. In general, template engines replace placeholders in files or strings with input values and return a formatted string. With template engines, we can create a number of templates that will be adapted dynamically based on the results of program candidate execution.

We consider two different designs of the instruction generation block:  $\text{INSTRUCT}^{\text{static}}$  and  $\text{INSTRUCT}^{\text{LLM}}$  shown in figure 3. Both types of blocks use failing I/O pairs from the validation set and `stderr` output of the candidate execution. In both blocks, if `stderr` is not empty, i.e., execution errors occur before getting the output to compare it with the expected output, the `stderr`-based template engine generates an instruction to fix the error mentioned in `stderr`. However, the blocks differ in the way they transform failing I/O pairs to generate instructions in case `stderr` is empty.

$\text{INSTRUCT}^{\text{static}}$  uses a fixed input template and substitutes placeholders for input and output with the corresponding strings of the first failing test case. We show the resulting instruction for an exemplar template in figure 3. By contrast,  $\text{INSTRUCT}^{\text{LLM}}$  uses the failing I/O pair in the LLM for text completion, thereby prompting the text LLM to produce the bug summary. An exemplar output of the code behavior template engine in figure 3 describes that

the code returns output  $O$  instead of expected output  $O_{\text{val}}$  for the failing test case with input string  $I_{\text{val}}$ . The LLM is then prompted to auto-complete this description of program behavior with the bug summary. The bug description is passed further to the next template engine and used as the debugging instruction, such as “Fix {bug summary}”.

### 2.5 Debug

The DEBUG block iterates over all programs in the population and uses the instruction written by INSTRUCT based on the results of EXECUTE to sample from  $p_{\text{debug}}$  (input, instr)  $N$  times to repair every candidate, setting input to the candidate solution and instruction to the output of INSTRUCT. The population of candidates is then replaced with the output of DEBUG.

### 2.6 Rank

Finally, the RANK block implements what is known in genetic programming as *parent selection* [39]. It ranks all programs in the candidate population by their score calculated in EXECUTE, keeps the top  $W$  programs, and removes the rest from the population.

### 2.7 Meaning of Hyperparameters

After evaluating a given candidate solution in EXECUTE, SEIDR supports two approaches to addressing the candidate’s flaws:

- Replace the candidate with another sample from the current population.
- Use INSTRUCT and DEBUG to repair the candidate.

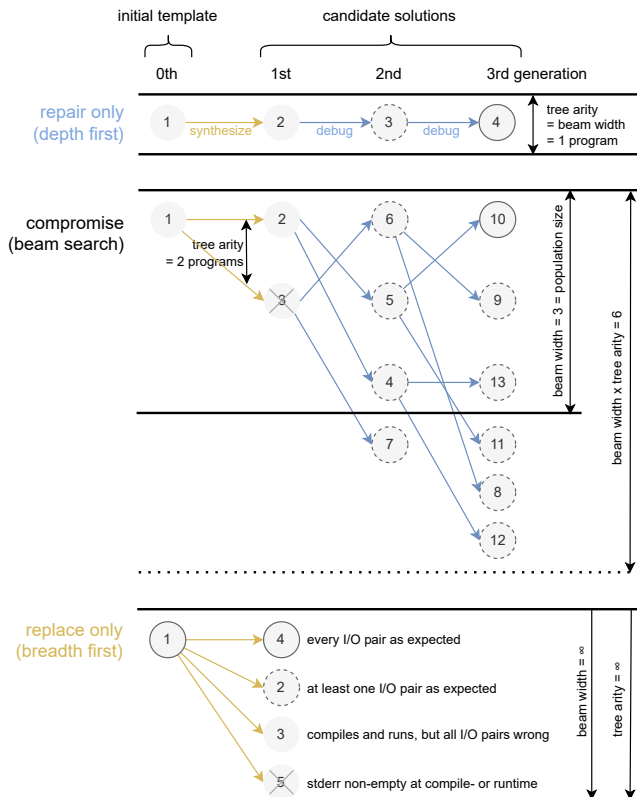
We refer to this problem as *repair-replace trade-off*, by analogy with production economics [40].

How does the choice of hyperparameters  $N$  and  $W$  influence the flow of SEIDR?  $N$  and  $W$  act as upper bounds on the *replace* option by limiting the size of the population. In the edge cases,  $N = W = 1$  corresponds to a repair-only process, while  $N = W = \infty$  corresponds to replace-only, see figure 4.

Observe that a mutation-only genetic algorithm with population size  $W$ , such as SEIDR, is equivalent to *local beam search* with beam width  $W$  on a  $N$ -ary tree [41, Section 4.1.4]. This corresponds to a known property of local beam search: it degenerates into depth-first search when  $W = 1$ , whereas setting  $W = \infty$  yields breadth-first search. Hence, we refer to  $N$  as *tree arity* and  $W$  as *beam width*.

## 3 RELATED WORK

The use of large language models for a program repair step within a program synthesis pipeline has been studied by Joshi et al. [42] and Gupta et al. [28], while a specific case of instruction-driven



**Figure 4: Repair-replace trade-off as a tree search problem.**

LLMs has been explored by Fan et al. [34]. The latter authors also compare instruction-driven LLMs to other Automated Program Repair (APR) strategies, and conclude that LLMs solve the task effectively. However, they do not consider the entire *Synthesize, Execute, Debug* framework, as we do in this work. By contrast, LLMs are usually set up to generate a fix from the first debug attempt and do not update failing patches iteratively.

Prompt generation for LLMs pre-trained on code is investigated in prompt engineering literature on a number of tasks, such as source code repair, type inference, code synthesis, and autocompletion [43–45]. Studies on program repair automation used prompts that contain code only, code with docstrings, and code with bug hints with the Codex model to test the repair capabilities of the LLM on the QuixBugs benchmark [46, 47]. The studies reported that bug localization hints were not helpful, whereas providing buggy code and the task summary was the most effective. ChatGPT was tested on QuixBugs in addition to Codex models as well [48]. Kuznia et al. [49] proposed to summarize task descriptions of competitive programming and interview programming tasks. Following guidance from these studies, we include I/O pairs and task descriptions as docstrings in addition to the function signature in our prompts.

## 4 EXPERIMENTAL SETUP

To explore the capabilities of SEIDR, we test the framework on the benchmark of problems for code competitions with different types of instructions for program candidate debugging, varied search strategies, and two languages, Python and C++. Our experiments

use the Program Synthesis Benchmark 2 (PSB2) for problem descriptions and tests to evaluate the proposed framework [50]. We compare the performance of programs synthesized with SEIDR to the PushGP genetic programming system with down-sampled lexicase selection [51]. During our empirical evaluation of SEIDR performance, we address the following research questions:

**RQ1. Repair-replace trade-off exploration:** What is the impact of using different tree search strategies in the autonomous programming setting? We experiment with four different tree arities in the tree search and study their impact on the number of resolved problems as well as the speed of obtaining solutions.

**RQ2. Prompt engineering:** What is the effect of using LLM-produced bug summaries compared to static instructions on the repair performance of automatically synthesized code? We test six static debug instructions that describe bug behavior based on violated requirements and five auto-generated debug prompts.

### 4.1 Data

PSB2 is a benchmark suite of 25 problems for program synthesis that resemble small real-world tasks. PSB2 was developed as a more realistic and challenging version of PSB1 [52], the latter consisting of textbook problems and is widely used in genetic programming [53]. The problems require different data structures and control flows to be used for effective solutions and are taken from sources, such as competitive programming platforms and educational courses. The problems have descriptions in English, as well as 1 million (M) tests for training and 1M testing-stage tests, including edge or corner cases that test the resulting program on complicated inputs. The tests are provided as I/O pairs and are distributed together with the problem descriptions as a PyPI package.<sup>4</sup>

In PSB1 [52], the training set consists of the edge test cases and is augmented by random test cases if the number of edge tests is not enough. The test set is formed by random test cases. This terminology is preserved in PSB2. However, we do not have a training or fine-tuning phase in our experiments, because the models are not made available for further training. Instead, we validate the framework with an existing pre-trained LLM for code and text as its parts. Therefore, we only have the validation and test phases. We will refer to training test cases in the PSB terminology as validation test cases in this study.

### 4.2 Repair-replace Trade-off Settings

As described in Section 2.7, the choice of beam width  $W$  and tree arity  $N$  define the repair-replace trade-off where higher  $W$  and  $N$  prioritize to repair over replace. We evaluate four options for these hyperparameters as shown in table 1.

**Table 1: Tree search hyperparameters.**

experiment	1	2	3	4
beam width $W$	1	10	100	$\infty$
tree arity $N$	1	10	100	$\infty$

Because we aim to compare tree search parameters, we fix one default debugging instruction and use the INSTRUCT<sup>static</sup> block. Moreover, we set the upper limit for the total number of generated program candidates to 1000 to limit the experimentation time.

<sup>4</sup> <https://pypi.org/project/psb2/>

Although some solutions may not be found within the hard limit, we assume<sup>5</sup> that 1000 program candidates form a sufficiently large search space for our experiments.  $W = N = \infty$  is achieved in implementation by setting  $W$  and  $N$  equal to the upper limit on the number of candidates, i. e. 1000. This setting ensures that a second generation of programs does not exist.

### 4.3 Prompting Strategies

The prompt for the LLM model  $p_{\text{debug}}(\text{input}, \text{instr})$  consists of the input for editing – candidate program generated so far – and a debug instruction to repair the candidate. We test SEIDR on 11 debug instructions to explore whether the use of the LLM for text completion  $p_{\text{text}}(\text{input})$  benefits the performance of our framework, as well as what effect different prompt phrases have on the debug process. We compare debug instructions that use neutral phrases with those that use more confident language and mimic experienced software developers, as well as shorter and longer instructions with different amounts of details about code behavior. To alleviate the effect of beam width and tree arity, we set  $N = W = 1$  and test the repair-only tree search strategy shown in figure 4. This strategy is used to gradually improve one program candidate throughout the search with no competing programs in the same generation.

The debug instructions are formulated as templates. The instructions describe the violated requirements in terms of the wrong output in a failing I/O test or summarize the bug to capture issues in code logic. We present debug instructions using the template engine format: the brackets  $\{\}$  denote that the placeholder in the brackets will be replaced with the value generated during execution,  $\{I_{\text{val}}\}$  and  $\{O_{\text{val}}\}$  stand for values failing I/O pair from the validation set. As shown in figure 3, the instruction to fix the execution errors, which abort the program before the resulting output is obtained, with `stderr` lines: `Fix {stderr}`. Static debug instructions that do not use LLM for bug summarization are as follows:

- S0 Make sure that  $\{I_{\text{val}}\} \rightarrow \{O_{\text{val}}\}$ ;
- S1 Make sure the code returns  $\{O_{\text{val}}\}$  for input  $\{I_{\text{val}}\}$ ;
- S2 Ensure that input  $\{I_{\text{val}}\}$  yields output  $\{O_{\text{val}}\}$ ;
- S3 Modify code to get  $\{O_{\text{val}}\}$  from  $\{I_{\text{val}}\}$ ;
- S4 Code must correspond instructions in comments and  $\{I_{\text{val}}\}$  must yield  $\{O_{\text{val}}\}$ ;
- S5 See comments in code and return  $\{O_{\text{val}}\}$  for input  $\{I_{\text{val}}\}$ .

The instruction S0 is the default instruction for tree arity experiments. It has an intuitive symbolic notation ( $\rightarrow$ ) instead of the word “return” or “yield”. In instructions S1–S3, we experiment with verbs and the order of output and input. Alternatively, in debug instructions S4–S5, we prompt the model to consider task description in the docstring in addition to providing the details of the failing I/O pair. Overall, instructions S0–S5 indicate the requirements to be met, but do not describe the current program’s behavior.

The second set of instructions use the LLM for text completion  $p_{\text{text}}(\text{input})$ . The instructions are designed so that the LLM is prompted to complete the sentence that should describe an error. In addition to validation I/O pairs, the following notation is used:  $\{O_p\}$  denotes the program candidate output for input  $\{I_{\text{val}}\}$ ,  $\{\text{task}\}$  is a placeholder for a problem description in English. Note that we do not include the incorrect output  $O_p$  of a generated candidate

program in debug instructions S0–S5, because it is recommended to avoid asking the model what not to do.<sup>6</sup> We denote the text completion LLM’s output as  $\{\text{bug}\}$  which should constitute the bug summary. Input templates to use LLM for bug description followed by debugging instruction templates (after “ $\rightarrow$ ”) are as follows:

- M6 The code should solve the following problem:  $\{\text{task}\}$ . The code must return  $\{O_{\text{val}}\}$  for input  $\{I_{\text{val}}\}$  but it returns  $\{O_p\}$ . Obviously, the error is that...  
 $\rightarrow$  Fix  $\{\text{bug}\}$ ;
- M7 The code should solve the following problem:  $\{\text{task}\}$ . The code must return  $\{O_{\text{val}}\}$  for input  $\{I_{\text{val}}\}$  but it returns  $\{O_p\}$ . The error is that...  
 $\rightarrow$  Fix  $\{\text{bug}\}$ ;
- M8 Problem description:  $\{\text{task}\}$ . The code must return  $\{O_{\text{val}}\}$  for input  $\{I_{\text{val}}\}$ , but it returns  $\{O_p\}$ . It is clear the error is that...  
 $\rightarrow$  Fix  $\{\text{bug}\}$ ;
- M9 There is clearly a bug in code, because the code returns  $\{O_p\}$  for input  $\{I_{\text{val}}\}$  but output  $\{O_{\text{val}}\}$  is expected. The bug is that...  
 $\rightarrow$  Fix  $\{\text{bug}\}$ ;
- M10 There is clearly a bug in code, because the code returns  $\{O_p\}$  for input  $\{I_{\text{val}}\}$  but output  $\{O_{\text{val}}\}$  is expected. The bug is that...  
 $\rightarrow$  Fix  $\{\text{bug}\}$  and modify the code to return  $\{O_{\text{val}}\}$  for input  $\{I_{\text{val}}\}$ .

Note that the text completion LLM does not use program candidates in its input, but only template inputs M6–M10 before the arrow.

Input M6 for the text completion LLM is used to evaluate the effect of the “confidence” sentiment on the bug summaries and debugging process. It is identical to input M7, except for the word “obviously”, which should reflect or confidence of the comment. Inputs M7 and M8 can be compared in the way the problem description is introduced, i.e., as a separate sentence similar to a spoken situation in prompt M7 or as a short title in M8.

Input templates M9 and M10 for text completion LLM are identical, but the instruction templates are different. Text completion inputs start with a “confidently” phrased statement that a bug is present in code. We include both the LLM output  $\{\text{bug}\}$  and description of the failing validation test case in debug instruction M10. Therefore, instructions M6–M9 rely mainly on the LLM output to summarize the bug, whereas instruction M10 also provides information about the expected output.

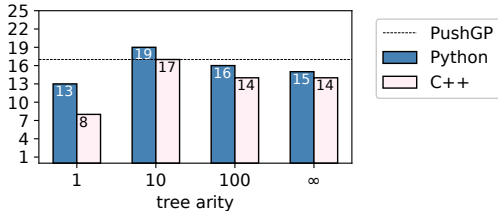
### 4.4 Performance Indicators

In our experiments, we compare the number of fully solved programs obtained using SEIDR with different values of hyperparameters. For a more detailed analysis of results, we use *test pass rate (TPR)* and *Excess Programs Generated (EPG)*. TPR reflects the percentage of fully passed test cases based on the exact match of program output and test output. The TPR metric is used for the final evaluation of generated programs and does not reflect partial passing of the I/O test as opposed to *score* in the RANK block.

DEBUG and EXECUTE blocks generate a number of programs that are replaced or repaired during the search for solution program. The number of programs generated before the first occurrence of the program that passes all validation test cases is referred to as EPG.

<sup>5</sup> This assumption is later confirmed in Section 5.1.

<sup>6</sup> <https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api>



**Figure 5: Number of solved PSB2 problems depending on the tree arity in tree search for the fixed prompt type S0.**

EPG is indicative of the computational cost of solving a problem distributed in terms of LLM inferences and program compilations and executions.

#### 4.5 Implementation Details

We use GPT-3 models pre-trained on code<sup>7</sup> and text<sup>8</sup> as LLMs in our framework. Specifically, we use Codex-edit (code-davinci-edit-001) as the LLM for draft programs  $p_{\text{synth}}$  and LLM for debugging  $p_{\text{debug}}$  and GPT-3 (text-davinci-003) for bug summarization via text completion with  $p_{\text{text}}$ . We ensure that the program candidates generated from the same parent program are different from each other by changing the temperature parameter of Codex-edit.

We use 2000 I/O pairs from the test split of PSB2 to evaluate the candidate program that has passed all the validation test cases during debugging. Due to repetitive calls to the EXECUTE block, we have to resolve the speed of testing versus precision trade-off while choosing the number of validation test pairs. We resolve the trade-off by fixing the validation set size at 100.

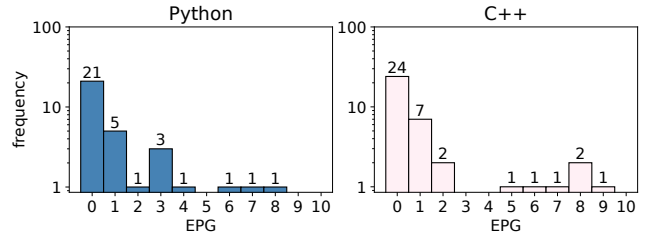
In the experiments with tree arity values, we set the limit to generate a maximum of 1000 program candidates during the search of the candidate that passes all validation tests. If we reach 1000 candidates and none of them passes all validation tests, we report the test pass rate for the last generated candidate. In the experiments with prompts, we set the limit of maximum generated programs to 5, because we search for the prompt that yields the fastest solution to exclude long searches and comply with the request rate limits.

## 5 RESULTS AND DISCUSSION

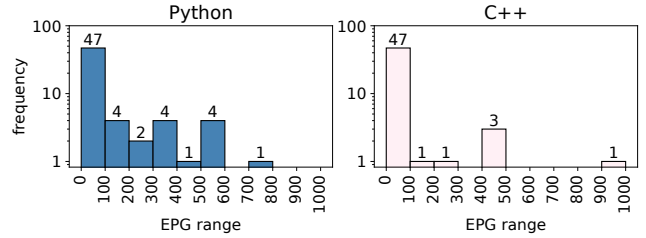
### 5.1 RQ1. Repair-replace Trade-off Exploration

We compare the number of solved problems in the experiments with tree arity of 1, 10, 100, and  $\infty$  and fixed debug instruction S0 in Python and C++ in figure 5. The results of SEIDR are compared to the baseline performance of PushGP on the PSB2 benchmark, which solves 17 out of 25 problems. Note that experiments with  $N = 1$  and  $N = \infty$  can be considered as ablation studies, where the replace option and repair option is turned off, correspondingly.

The results highlight the benefit of compromise strategies with tree arity of 10 and 100 over repair-only ( $N = 1$ ) and replace-only ( $N = \infty$ ) strategies. The repair-only scheme is outperformed by other strategies. We explain the poor performance of repair-only strategy by the fact that the search space is under-explored. Specifically, replace scenario ensures the LLM for debugging represented by Codex-edit in our experiments generates different updates of



**(a)  $0 \leq \text{EPG} \leq 10$  with step 1.**



**(b)  $0 \leq \text{EPG} \leq 1000$  with step 100.**

**Figure 6: Distribution of the number of generated programs during each problem-solving attempt in the experiments with different tree arities where a problem solution is found.**

program candidates using variable temperature. The probability of finding a better fix is higher when more alternatives are generated to update the draft program at  $N > 1$  compared to  $N = 1$ . The search strategy with  $N = 10$  yields the best results: it performs on par with PushGP for C++ and outperforms the baseline during Python program synthesis by +2 problems resulting in a total of 19 programs that pass all test cases. The results imply that generating a moderate number of programs in parallel during the DEBUG step works better than the policies in which more updates are generated for each program (100 or 1000) or only one program is updated iteratively.

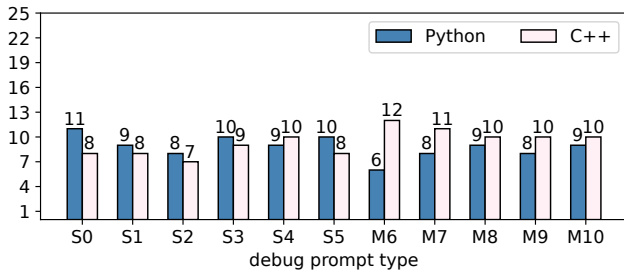
We present the analogy of the solution speed for all four arities and fixed default debug instruction in figure 6. In detail, we show the distribution of EPG values in all experiments to explore how many candidate updates are generated before the solution is found. We zoom in to the cases with solutions found with up to the first 10 program candidates in figure 6a and show the EPG distribution with the step of 100 candidates in figure 6b.

Out of 100 experiments for each language, in 21–24% of runs in Python and C++, the draft program is already the solution (EPG=0). For 19–32% of experiments, the solution is found after discarding 5 candidates. Around half of experiments do not generate more than 100 programs. However, 5 problems are solved with more than 500 generated programs in Python and 1 problem in C++ (with  $N = 10$ ). The results imply that the first steps in the update of the draft program are crucial for solving the problem. The chances of solving the problem on the later stages of the search, such as after 100 programs have been generated, are low. This confirms our initial assumption in Section 4.2 that 1000 programs are sufficient.

**Answer to RQ1.** SEIDR outperforms the PushGP baseline on PSB2 in Python and performs on par with it in C++ experiments

<sup>7</sup> <https://platform.openai.com/docs/guides/code/editing-code>

<sup>8</sup> <https://platform.openai.com/docs/models/gpt-3>



**Figure 7: Number of solved PSB2 problems depending on the instruction choice for the fixed tree arity of 1.**

with tree arity of 10. Search strategies with tree arity larger than one benefit from the replace possibility of the SEIDR framework as a consequence of using variable temperature for Codex-edit. The repair component is also crucial for the framework because the replace-only search policy (with tree arity of  $\infty$ ) performs worse than the policies that alternate between replace and repair during program update (with tree arity of 10 or 100).

## 5.2 RQ2. Prompt Engineering

We report the number of solved problems for different static and GPT-assisted debug instructions in figure 7. Because debug instructions are parts of prompts for LLMs and the program candidate format does not change, we will use the term *prompt* during the analysis of experiment results with different instructions. Overall, the performance of the framework is robust to the debug prompt choice, both with LLM-generated and static templates. The number of solved problems differs for Python and C++ in our experiments.

For C++, all debug prompts except S2 result in the same or higher performance than the instruction S0 which is used in the repair-replace trade-off experiments. The debug instruction S2 contains the verbs “yield” and “ensure” which are probably rarely used in code documentation. The best debug instruction for C++ is the LLM-assisted template M6 containing the word “obviously”, which should indicate the confidence of the author of bug summary whom GPT-3 should mimic during autocompletion.

Python programs do not show the same effect during experiments with different prompts. The overall performance drops in comparison with using the prompt S0. By limiting the total number of generated programs from 1000 to 5 in the current set of experiments, we lose 2 problem solutions in Python with S0. The prompt that results in the best performance in C++ for the EPG limit of 5 corresponds to the worst performance in Python. This result can occur due to the small tree arity and low variability of debugging updates of the initial draft. Another reason is that the GPT-3 summary of bugs may not point to logical errors. The model for text autocompletion frequently outputs bug summaries that mention “the code is not accepting the input correctly.” Note that such bug summary appears in other debug prompts, too.

To analyze the effect of using different prompts on a problem level, we present a heatmap of EPG for all 25 problems in figure 8. We add the values of test pass rate in numbers or signs and show EPG in color. Empty cells denote that the search halts due to OpenAI exceptions, such as `APIError`.<sup>9</sup> In addition, if the framework halts

before max programs attempts (light-blue cells with a “-”), it is due to the input length limit of the underlying LLM  $p_{\text{debug}}$ , i.e., the generated code is too long and does not fit as input to the LLM.

Some problems are solved with all prompts, while other problems are solved with only a subset of prompts, solved partially, or not solved at all. A number of problems are solved with all or the majority of prompts in both languages, such as `basement`, `fizz-buzz`, `paired-digits`, and `twitter`. Other problems pass all tests in only one of the languages, such as `luhn`, `vector-distance`, `fuel-cost`, or `substitution-cipher`. Most of the solved problems are generated as the first draft or within 1–2 debug steps. However, some problems pass 90% of test cases at the fifth step, such as `substitution-cipher` in Python with prompts S4 and M8 or `shopping-list` in C++ with prompts S0, S1, S5 and M7. These runs are likely to be updated with the fully correct programs in the following several steps, according to the results in section 5.1, but the experiments are stopped for the fairness of inter-prompt comparison. Alternatively, conducting the prompt engineering experiment with 1000 max programs would have shown what prompts are beneficial for solving the problems in the long run and can be interesting for future work.

The most interesting cases concern the problems that are solved only with LLM bug summaries or only with static prompts. For example, the `gcd` problem is solved only with prompts M6–M10 in C++ and is not solved with either of S0–S5. A similar result is obtained for `spin-words` and `coin-sums` in C++. In Python, we observe only the cases where solutions are obtained with static prompts and are not obtained with GPT-assisted prompts, e.g., for `find-pair`, `camel-case`. In addition, several prompts work well from both S and M categories as for `gcd` in Python.

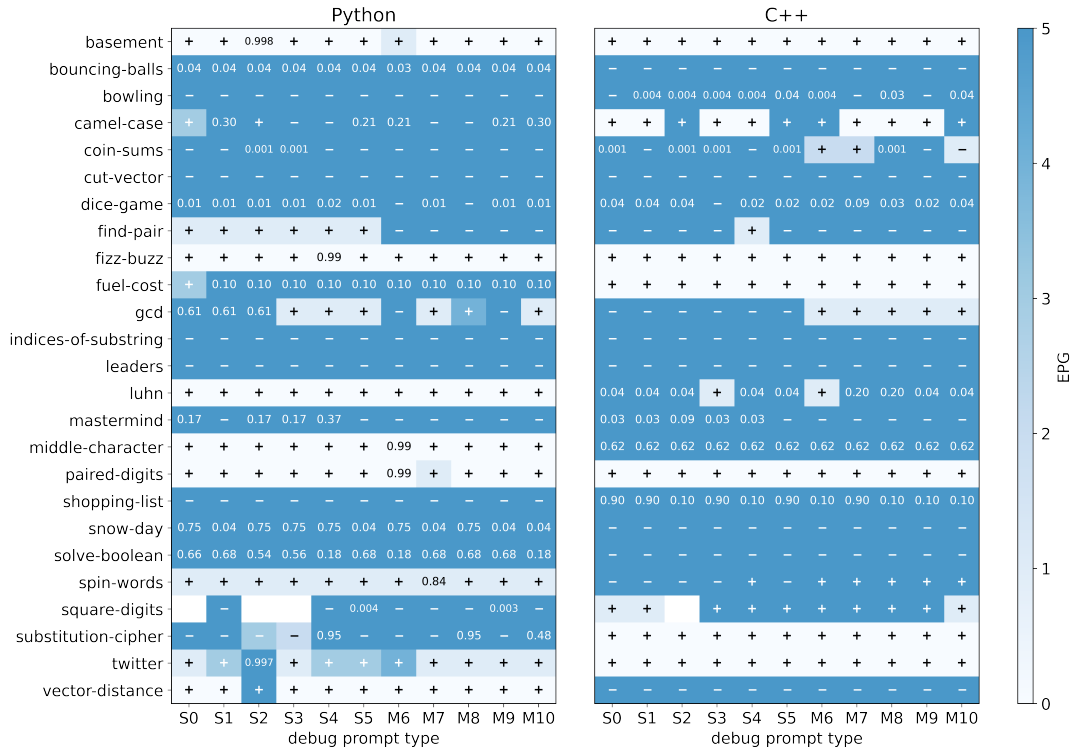
**Answer to RQ2.** Program synthesis in C++ with SEIDR achieves better performance in the repair-only setting with both GPT-assisted prompts that summarize bugs in code and static templates which describe failing I/O cases. The best-performing C++ instruction is obtained with GPT-3 for text completion that contains the word “obviously”. Results differ for PSB2 solutions in Python: the static prompt template S0 results in the best performance. Overall, SEIDR performance is stable with different debugging prompts submitted to Codex-edit.

## 5.3 Threats to Validity

External threats to validity concern SEIDR performance on different benchmarks and the use of other language models than the tested ones. Specifically, PSB2 contains competitive programming tasks which require smaller functions to be generated than production-scale software. We plan to extend our experiments in future work to explore the generalizability of results to other benchmarks.

Internal threats relate to the implementation. We use PSB2, which has corner case tests in the training set and test regular cases in the test set. To ensure a fair comparison with other studies on PSB2, we evaluate and report results on the provided test set of PSB2 which risks that the synthesized programs do not pass some of the training cases. Large models for code editing and text completion used in this study are nondeterministic, which impacts results. Due to prohibitive model inference costs, each experiment was only run once. However, our temperature sampling procedure

<sup>9</sup> <https://platform.openai.com/docs/guides/error-codes/python-library-error-types>



**Figure 8: Number of excess programs generated (in color) and test pass rate (as numbers) depending on the type of debug prompt. Higher EPG values are shown in darker shades than low EPG. We denote solved problems with “+” (test pass rate = 1), unsolved problems with “-” (test pass rate = 0), and show the test pass rate for partially solved problems.**

described in section 2.2 reduces this stochasticity significantly, especially for low-EPG results. As with other language models, Codex is a black-box model and may generate malicious code [54]. The Codex model was pre-trained on an unbalanced dataset across programming languages [9]. Thus, the results can be skewed towards high performance in popular programming languages.

## 6 CONCLUSION

In this study, we propose the SEIDR framework to solve the challenge of fully autonomous programming. We augment the program synthesis procedure based on the large language models for code generation from templates and textual instructions with the repair block. The repair block consists of the tree search across the program candidates generated by a large language model for code. The LLM used for code repair takes imperfect program candidates and instructions for their improvement as prompts. The instructions are obtained from both static templates with failing test case descriptions and templates with auto-generated bug summaries by a text completion language model. We explore 11 prompting strategies and the repair-replace trade-off of updating the draft program.

**Contributions:** We test SEIDR with the Codex-edit as the model for draft program synthesis and debugging in Python and C++ on the PSB2 benchmark. In our experiments, SEIDR outperforms the PushGP baseline and achieves the state-of-the-art result with 19

solved problems out of 25. It requires under 1000 program executions to solve them, in stark contrast to billions<sup>10</sup> of executions in PushGP, making it feasible in the areas with costly testing, such as robotics. Investigation of the repair-replace trade-off shows that SEIDR with tree arity of 10 outperforms both the replace-only strategy and the repair-only approach. Our prompt engineering study shows that bug summaries generated with “confidence indicators”, such as “obviously”, improve the performance of SEIDR during C++ code synthesis. Overall, our framework shows low performance variability with different prompts, which indicates its robustness. **Future work:** To study the generalizability of the SEIDR framework, we plan to expand the experiments to the competitive programming dataset of AlphaCode [10] and QuixBugs [46], as well as experimenting with ranking strategies, such as lexicase selection.

## DATA AVAILABILITY

The code and results are made available via Zenodo.<sup>11</sup> Note that OpenAI discontinued the Codex API on March 23, 2023, and suggests using the GPT-3.5-Turbo API instead.

<sup>10</sup>A problem is considered “solved” by PushGP if at least 1 of 100 runs, each with a limit of 60 million programs, was successful.

<sup>11</sup> <https://doi.org/10.5281/zenodo.7837282>



## ACKNOWLEDGMENTS

The work presented in this paper was supported by the European Commission through Horizon 2020 grant 812882, and by the Research Council of Norway through the secureIT project (#288787). The empirical evaluation made use of the Experimental Infrastructure for Exploration of Exascale Computing (eX3), supported by the Research Council of Norway through grant #270053.

## REFERENCES

- [1] Z. Manna and R. J. Waldinger. “Toward Automatic Program Synthesis.” In: *Communications of the ACM* 14.3 (1971), pp. 151–165. doi: 10/bhgh4z.
- [2] O. Bastani, J. P. Inala, and A. Solar-Lezama. “Interpretable, Verifiable, and Robust Reinforcement Learning via Program Synthesis.” In: *xxAI - Beyond Explainable AI: International Workshop, Held in Conjunction with ICML 2020, July 18, 2020, Vienna, Austria, Revised and Extended Papers*. Ed. by A. Holzinger, R. Goebel, R. Fong, T. Moon, K.-R. Müller, and W. Samek. Lecture Notes in Computer Science. Springer, 2022, pp. 207–228. doi: 10/j5zn.
- [3] S. Dhar, J. Guo, J. Liu, S. Tripathi, U. Kurup, and M. Shah. “A Survey of On-Device Machine Learning: An Algorithms and Learning Theory Perspective.” In: *ACM Transactions on Internet of Things* 2.3 (2021), pp. 1–49. doi: 10/gnxq57.
- [4] T. M. Connolly, M. Soflano, and P. Papadopoulos. “Systematic Literature Review: XAI and Clinical Decision Support.” In: *Diverse Perspectives and State-of-the-Art Approaches to the Utilization of Data-Driven Clinical Decision Support Systems*. IGI Global, 2023, pp. 161–188. doi: 10/j5zp.
- [5] Y. Jia, J. McDermid, T. Lawton, and I. Habli. “The Role of Explainability in Assuring Safety of Machine Learning in Healthcare.” In: *IEEE Transactions on Emerging Topics in Computing* 10.4 (Oct. 2022), pp. 1746–1760. doi: 10/grm36.
- [6] Z. Manna and R. Waldinger. “Fundamentals of Deductive Program Synthesis.” In: *IEEE Transactions on Software Engineering* 18.8 (1992), pp. 674–704. doi: 10/ddbdf.
- [7] R. Alur et al. “Syntax-Guided Synthesis.” In: *Dependable Software Systems Engineering* (2015), pp. 1–25. doi: 10/grb6m9.
- [8] M. T. Ahvanooy, Q. Li, M. Wu, and S. Wang. “A Survey of Genetic Programming and Its Applications.” In: *KSII Transactions on Internet and Information Systems (TIIS)* 13.4 (2019), pp. 1765–1794.
- [9] M. Chen et al. *Evaluating Large Language Models Trained on Code*. July 2021. arXiv: 2107.03374.
- [10] Y. Li et al. “Competition-Level Code Generation with AlphaCode.” In: *Science* 378.6624 (Dec. 2022), pp. 1092–1097. doi: 10/grggxf.
- [11] S. Ren et al. *CodeBLEU: A Method for Automatic Evaluation of Code Synthesis*. Sept. 2020. arXiv: 2009.10297.
- [12] R. Bavishi, H. Joshi, J. Cambroner, A. Fariha, S. Gulwani, V. Le, I. Radicek, and A. Tiwari. “Neurosymbolic Repair for Low-Code Formula Languages.” In: *OOPSLA*. Dec. 2022.
- [13] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. “A Survey of Machine Learning for Big Code and Naturalness.” In: *ACM Computing Surveys* 51.4 (July 2018), 81:1–81:37. doi: 10/gffkxm.
- [14] X. Chen, D. Song, and Y. Tian. “Latent Execution for Neural Program Synthesis Beyond Domain-Specific Languages.” In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 22196–22208.
- [15] O. Polozov and S. Gulwani. “FlashMeta: A Framework for Inductive Program Synthesis.” In: *OOPSLA*. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 107–126. doi: 10/grb6mm.
- [16] V. Liventsev, A. Härmä, and M. Petković. *BF++: A Language for General-Purpose Program Synthesis*. Jan. 2021. arXiv: 2101.09571.
- [17] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu. “A Survey of Deep Learning Techniques for Autonomous Driving.” In: *Journal of Field Robotics* 37.3 (2020), pp. 362–386. doi: 10/gg9ztb.
- [18] M. Marcano, S. Díaz, J. Pérez, and E. Irigoyen. “A Review of Shared Control for Automated Vehicles: Theory and Applications.” In: *IEEE Transactions on Human-Machine Systems* 50.6 (2020), pp. 475–491. doi: 10/gj8zjj.
- [19] S. Lu et al. “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation.” In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. Dec. 2021.
- [20] C. Niu, C. Li, V. Ng, and B. Luo. *CrossCodeBench: Benchmarking Cross-Task Generalization of Source Code Models*. Feb. 2023. arXiv: 2302.04030.
- [21] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. June 2020. arXiv: 1909.09436.
- [22] B. Roziere, M.-A. Lachaux, G. Lample, and L. Chausson. “Unsupervised Translation of Programming Languages.” In: *34th Conference on Neural Information Processing Systems*. Vancouver, Canada, 2020.
- [23] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. “A Review-Based Comparative Study of Bad Smell Detection Tools.” In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’16. New York, NY, USA: Association for Computing Machinery, June 2016, pp. 1–12. doi: 10/ggfv7h.
- [24] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. “Deep Learning Based Vulnerability Detection: Are We There Yet?” In: *IEEE Transactions on Software Engineering* 48.9 (Sept. 2022), pp. 3280–3296. doi: 10/gk52qr.
- [25] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. “Genetic Improvement of Software: A Comprehensive Survey.” In: *IEEE Transactions on Evolutionary Computation* 22.3 (June 2018), pp. 415–432. doi: 10/gdrjb7.
- [26] C. Le Goues, M. Pradel, and A. Roychoudhury. “Automated Program Repair.” In: *Communications of the ACM* 62.12 (Nov. 2019), pp. 56–65. doi: 10/gkgf29.
- [27] *TIOBE Index*. <https://www.tiobe.com/tiobe-index/>.
- [28] K. Gupta, P. E. Christensen, X. Chen, and D. Song. “Synthesize, Execute and Debug: Learning to Repair for Neural Program Synthesis.” In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 17685–17695.
- [29] S. Iyer, I. Konstantas, A. Cheung, and L. Zettlemoyer. *Mapping Language to Code in Programmatic Context*. Aug. 2018. arXiv: 1808.09588.
- [30] D. C. Halbert. “Programming by Example.” PhD thesis. University of California, Berkeley, 1984.
- [31] S. Gulwani. *Programming by Examples (and Its Applications in Data Wrangling)*. Tech. rep. Redmond, WA, USA: Microsoft Corporation, 2016, p. 22.
- [32] M. Zavershynskiy, A. Skidanov, and I. Polosukhin. *NAPS: Natural Program Synthesis Dataset*. 2018. arXiv: 1807.0316.
- [33] L. Ouyang et al. *Training Language Models to Follow Instructions with Human Feedback*. Mar. 2022. arXiv: 2203.02155.
- [34] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan. *Automated Repair of Programs from Large Language Models*. Jan. 2023. arXiv: 2205.10583.
- [35] J. Blain. *Nine Worlds of Seid-magic: Ecstasy and Neo-shamanism in North European Paganism*. Routledge, 2002.
- [36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. “Attention Is All You Need.”

- In: *International Conference on Neural Information Processing Systems (NeurIPS)*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 5998–6008.
- [37] *OpenAI API*. <https://platform.openai.com>.
- [38] S. de Bruin, V. Liventsev, and M. Petković. *Autoencoders as Tools for Program Synthesis*. Sept. 2021. arXiv: 2108.07129.
- [39] J. R. Koza. *Genetic Programming II*. Vol. 17. MIT Press, 1994.
- [40] N. Jack and F. Van der Duyn Schouten. “Optimal Repair–Replace Strategies for a Warranted Product.” In: *International Journal of Production Economics* 67.1 (Aug. 2000), pp. 95–100. doi: 10/cfzj7f.
- [41] S. J. Russell. *Artificial Intelligence a Modern Approach*. Pearson Education, Inc., 2010.
- [42] H. Joshi, J. Cambronero, S. Gulwani, V. Le, I. Radicek, and G. Verbruggen. *Repair Is Nearly Generation: Multilingual Program Repair with LLMs*. Sept. 2022. arXiv: 2208.11640.
- [43] D. Shrivastava, H. Larochelle, and D. Tarlow. *Repository-Level Prompt Generation for Large Language Models of Code*. Oct. 2022. arXiv: 2206.12839.
- [44] Q. Huang, Z. Yuan, Z. Xing, X. Xu, L. Zhu, and Q. Lu. *Prompt-Tuned Code Language Model as a Neural Knowledge Base for Type Inference in Statically-Typed Partial Code*. Aug. 2022. arXiv: 2208.05361.
- [45] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce. *Fixing Hardware Security Bugs with Large Language Models*. Feb. 2023. arXiv: 2302.01215.
- [46] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. “QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge.” In: *Companion of the SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. Vancouver BC Canada: ACM, Oct. 2017, pp. 55–56. doi: 10/gf8nmn.
- [47] J. A. Prenner, H. Babii, and R. Robbes. “Can OpenAI’s Codex Fix Bugs?: An Evaluation on QuixBugs.” In: *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*. May 2022, pp. 69–75. doi: 10/grpcnx.
- [48] D. Sobania, M. Briesch, C. Hanna, and J. Petke. *An Analysis of the Automatic Bug Fixing Performance of ChatGPT*. Jan. 2023. arXiv: 2301.08653.
- [49] K. Kuznia, S. Mishra, M. Parmar, and C. Baral. *Less Is More: Summary of Long Instructions Is Better for Program Synthesis*. Oct. 2022. arXiv: 2203.08597.
- [50] T. Helmuth and P. Kelly. “Applying Genetic Programming to PSB2: The next Generation Program Synthesis Benchmark Suite.” In: *Genetic Programming and Evolvable Machines* 23.3 (Sept. 2022), pp. 375–404. doi: 10/gq5gjq.
- [51] T. Helmuth and L. Spector. “Problem-Solving Benefits of Down-Sampled Lexicase Selection.” In: *Artificial Life* 27.3–4 (Mar. 2022), pp. 183–203. doi: 10/grrnj7.
- [52] T. Helmuth and L. Spector. “General Program Synthesis Benchmark Suite.” In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. Madrid Spain: ACM, July 2015, pp. 1039–1046. doi: 10/ghsn5b.
- [53] D. Sobania, M. Briesch, and F. Rothlauf. “Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of Github Copilot and Genetic Programming.” In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Boston Massachusetts: ACM, July 2022, pp. 1019–1027. doi: 10/gq5gjp.
- [54] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. *Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions*. Dec. 2021. arXiv: 2108.09293.