Repair Is Nearly Generation: Multilingual Program Repair with LLMs

Harshit Joshi[1], **José Cambronero Sanchez**[2*], **Sumit Gulwani**[2*], **Vu Le**[2*], **Ivan Radiček**[3*], **Gust Verbruggen**[4*]

[1] Microsoft, India
[2] Microsoft, USA
[3] Microsoft, Croatia
[4] Microsoft, Belgium
{t-hjoshi, jcambronero, sumitg, levu, ivradice, gverbruggen}@microsoft.com

Abstract

Most programmers make mistakes when writing code. Some of these mistakes are small and require few edits to the original program – a class of errors recently termed *last mile mistakes*. These errors break the flow for experienced developers and can stump novice programmers. Existing automated repair techniques targeting this class of errors are language-specific and do not easily carry over to new languages. Transferring symbolic approaches requires substantial engineering and neural approaches require data and retraining. We introduce RING, a multilingual repair engine powered by a large language model trained on code (LLMC) such as Codex. Such a multilingual engine enables a *flipped model* for programming assistance, one where the programmer writes code and the AI assistance suggests fixes, compared to traditional code suggestion technology. Taking inspiration from the way programmers manually fix bugs, we show that a prompt-based strategy that conceptualizes repair as localization, transformation, and candidate ranking, can successfully repair programs in multiple languages with minimal effort. We present the first results for such a multilingual repair engine by evaluating on 6 different languages and comparing performance to language-specific repair engines. We show that RING can outperform language-specific repair engines for three of these languages.

Introduction

The number of people writing code across different languages has steadily grown (Bureau of Labor Statistics 2022) and ranges from novices to experts. Regardless of their experience level, programmers can make mistakes when writing code. Program errors can range from those that are easy to spot and fix, to those that require substantial application knowledge and may be very subtle logical bugs. Even simple mistakes, such as syntax errors that require a relatively small edit and may be apparent to a programming expert, can be frustrating for novice programmers. Moreover they can slow down the workflow of more experienced programmers (Wexelblat 1976; Murphy et al. 2008; Altadmri and Brown 2015; Drosos, Guo, and Parnin 2017).

One way to help programmers who encounter these small mistakes is by using automated program repair (APR). These methods take a faulty program and a specification of correctness as input, and return as output a fixed version of the program that conforms to the specification.

Recent work (Bavishi et al. 2022) has introduced the term *last-mile repairs* to broadly describe the class of repairs where the original program is a small edit distance away from the correct program. In this definition, program correctness can be checked without substantial additional context—a parser and a type checker suffice. A quick search on most programming help forums reveals a large number of questions for such errors. For example, as of August 2022, there are over 15K posts on StackOverflow tagged with Python and SyntaxError.

Existing work has explored performing these kind of repairs automatically. Symbolic systems, such as Grmtools (Diekmann and Tratt 2020), typically build on error-recovery mechanisms in parsers to enumerate local edits that can resolve errors raised during parsing. Symbolic systems typically restrict the search space to avoid state explosions and they cannot easily encode properties such as the likelihood of particular repair candidates being correct or not.

More recently, neural approaches have been successfully applied to repairing syntax and diagnostics errors. For example, Dr. Repair (Yasunaga and Liang 2020), BIFI (Yasunaga and Liang 2021), and TFix (Berabi et al. 2021) use transformer architectures to produce repairs for C compilation errors, Python syntax errors, and JavaScript linter diagnostics, respectively. Some systems, such as LaMirage (Bavishi et al. 2022), have also combined symbolic and neural components to successfully repair broken programs in low-code languages such as Excel and Power Fx.

Unfortunately, all these systems share a key drawback: they require substantial engineering (symbolic) or additional data and training (neural) to adapt to new languages. In this paper, we propose a single repair engine, that leverages a large language model trained on code (LLMC) to perform multilingual repair. We select Codex by OpenAI as the LLMC.

Our system, RING, shows that **r**epair **i**s **n**early **g**eneration and exploits Codex's few-shot learning capabilities (Bareiß et al. 2022; Drori et al. 2022) to perform multilingual program repair. To do this effectively, we break down program repair into the same three phases as symbolic automated program repair systems: fault localization, code transformation, and candidate ranking (Goues, Pradel, and Roychoudhury 2019;

*Listed in alphabetical order

Liu et al. 2021; Bavishi et al. 2022). We show how each stage can be addressed with minimal effort by emulating what a developer would do and using this intuition to design prompts for an LLMC.

We evaluate RING on six languages: Excel, Power Fx, Python, JavaScript, C and PowerShell. Our results show that RING repairs significantly more programs than a language-specific repair engine for three languages and shows competitive results for another two languages. We evaluate the effectiveness of our design choices for each of the three stages of repair. Additionally, we identify possible directions for improvement based on our results, such as language-specific ranking and iterative querying with Codex.

Jointly, these results provide the first evidence that an LLMC can enable multilingual repair with the same or better performance than methods designed for a single language. In contrast to other AI-assisted code editing features, such as code completion, this advance opens up the possibility of a *flipped interaction model* where the user writes code and the AI assistant performs the fixing.

In summary, we make the following contributions:

- We present an LLMC-based approach to multilingual repair that enables a flipped interaction model for AI-assisted programming in which the user writes code and the assistant suggests fixes for last-mile mistakes.
- We implement our approach in the RING system, which employs compiler (or diagnostic) messages, smart few-shot selection, and ranking of repair candidates to perform repair across varying languages.
- We perform an extensive evaluation across six different languages, showing that multilingual repair with LLMCs is viable and can compete with or outperform language-specific repair engines.
- We introduce PowerShell commands as a new application for last-mile repair and collect a benchmark set of 200 PowerShell commands from StackOverflow, which we also release for future research[1].

Related Work

Automated Program Repair Finding and fixing bugs is challenging and tedious, even for language experts (Zhong and Su 2015). The software engineering community has built Automated Program Repair (APR) tools (Arcuri 2008) to reduce the time and costs associated with debugging. The premise of APR has since grown into a substantial research domain across different languages, classes of bugs, and use cases (Gazzola, Micucci, and Mariani 2019).

Early approaches for APR were symbolic and attempted to fix programs automatically by enumerating repair candidates from templates (Debroy and Wong 2010), crafting heuristics (Qi et al. 2014), and using program synthesis (Nguyen et al. 2013). Although these systems can provide strong guarantees for the generated code, they are strongly tied to their domain language. Moreover, symbolic systems are restrictive in their scope, failing to repair programs that the corresponding language compiler cannot process to at least some extent. On the other hand, building on the recent advances in natural language processing, neural methods have shown promise in learning program repairs. Researchers have studied automatically correcting programs in different settings, including introductory programming assignments (Pu et al. 2016; Parihar et al. 2017; Ahmed et al. 2018). For example, DeepFix (Gupta et al. 2017) and SampleFix (Hajipour, Bhattacharyya, and Fritz 2020) use sequence-based deep learning models to fix broken C code written by students. However, these neural models are not as powerful as LLMCs like Codex.

SynFix (Ahmed, Ledesma, and Devanbu 2021), Dr. Repair (Yasunaga and Liang 2020), and TFix (Berabi et al. 2021) leverage compiler diagnostics for Java, C, and JavaScript, respectively, but require a substantial amount of training and data, failing to generalize across languages. Although (Bavishi et al. 2022) tries to bridge the gap between neural and symbolic approaches, their approach requires language specialization (symbolic parser) and large-scale data (neural localizer and ranker). In contrast, RING uses a powerful LLMC, Codex, capable of generating multilingual code while guiding repair through readily available prompt-design strategies.

Large Language Models The advent of Large Language Models (LLM) trained on code and natural language shows promise for code understanding and generation results. Autoregressive models (Shannon 1948; Radford et al. 2018), such as Codex (Chen et al. 2021), are trained to predict the next token, given the past token context, over enormous corpora. However, training LLMs is technically challenging and expensive. They require a large dataset for each fine-tuning task and parallel training on multiple GPUs (Bommasani et al. 2021) to scale. An exciting aspect of LLMs is the zero-shot and few-shot learning paradigm for adapting to tasks on-the-fly (Brown et al. 2020; Chowdhury, Zhuang, and Wang 2022).

Prenner and Robbes (2021) evaluate Codex's ability to fix 80 bugs in Python and Java using such in-context learning abilities. They manually provide buggy lines for each program and use *fixed* few shot examples in the prompt. In contrast, our paper discusses various strategies to build the prompt, and performs a more extensive study with larger datasets over more languages.

Approach

Figure 1 shows the architecture of RING. We divide the task of fixing bugs into three stages: fault localization, program transformation and candidate ranking. Each stage is based on the intuition for how developers might approach such a stage manually. In the following subsections, we show how to address each stage using an LLMC.

We illustrate our approach using a running example – shown in Figure 2 – drawn from the BIFI (Yasunaga and Liang 2021) dataset. The user has incorrectly used tuple notation in the function signature (highlighted in pink). This syntax for unpacking tuples in function signatures was supported in Python 2. In Python 3, it raises a syntax error[2] with very little detail on the underlying issue. This example high-

[1] https://github.com/microsoft/prose-benchmarks/

[2] https://peps.python.org/pep-3113/

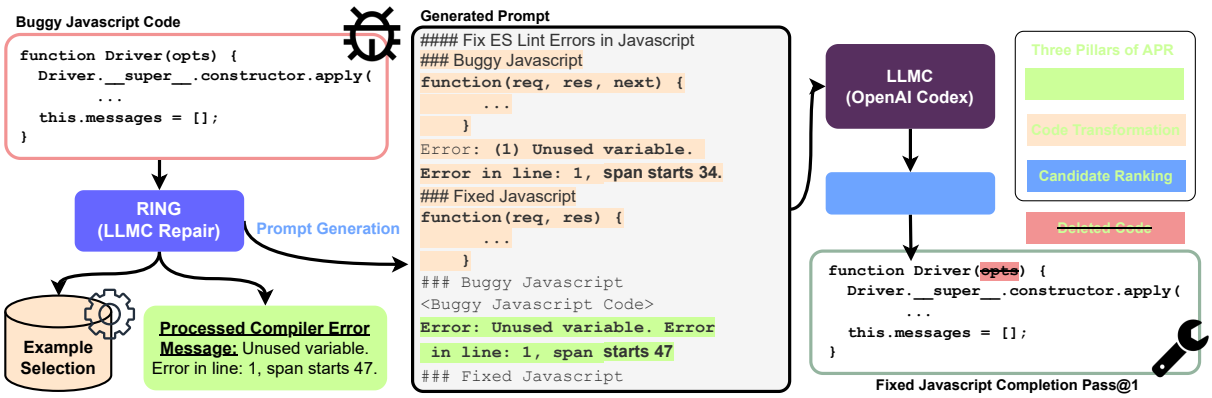


Figure 1: RING, powered by a Large Language Model trained on Code (LLMC), performs multi-lingual program repair. RING obtains fault localization information from error messages and leverages LLMC's few shot capabilities for code transformation through example selection, forming the prompt. Finally, a simple, yet effective, technique is used for ranking repair candidates.

```
def boundary_difference_power(graph,
    (orig_image, sigma, spacing)):
  orig_image = scipy.asarray(orig_image)
  def boundary_term_division(i):
    i = 1. /(i + 1)
    i = scipy.power(i, sigma)
    i[i <= 0] = sys.float_info.min
    return i
  __skeleton_difference(graph,
        orig_image,
        boundary_term_division)
```

Figure 2: A real Python 3 syntax error from the BIFI dataset. The highlighted code uses tuple parameter unpacking syntax, which was valid in Python 2 but removed from Python 3. All listings are simplified for presentation clarity and brevity.

lights that errors can also be introduced as languages evolve. RING fixes this mistake without additional user intervention.

Fault Localization through Language Tooling

As a first step towards debugging, a programmer typically locates the cause of the bug. For most modern languages, locating syntactic mistakes and some semantic errors, such as type errors, is aided by tools like the compiler, static analyzers, or linters. Following this intuition, we include a preprocessed error message produced by the compiler or other static analyzers. We normalize this message to enforce consistency across languages. Figure 3 shows this prompt variant for our running example, where the highlighting corresponds to our prepared syntax error message. For languages where the error messaging may not be precise, particularly with regards to the error location reported, we found that a simple abstraction that removes the reported error location but preserves the error text worked well – we discuss how to create such an abstracted message in our discussion section.

```
### Buggy Python
def boundary_difference_power(graph,
    (orig_image, sigma, spacing)):
...
Error: (1) invalid syntax. Error in
line: 2 span starts 4 and ends 32.
```

Figure 3: To aid fault localization, we include a detailed compiler error message with line/column span information. We prepare uniform messages across languages by extracting details from the corresponding language compiler/analyzer.

Code Transformation through Few-shot Learning

Once a developer has identified the location of a mistake, they must now apply an appropriate transformation—a sequence of edits—to the original source code at this location. Most developers accumulate experience in the type of transformations needed to resolve particular errors over time. Additionally, when novices encounter an unfamiliar mistake, they often search for examples of similar buggy/correct pairs that can inform their own transformation.

It has been shown that LLMs are capable of few-shot learning—the ability to learn from a few examples of the intended task—by adding related examples of the task to the prompt (Brown et al. 2020; Poesia et al. 2022). Given examples of transformations that repair programs, we exploit this capability in RING to address the code transformation stage. The main challenge is selecting relevant examples that are related to the mistake made by the developer.

Following the intuition that programs with similar mistakes have similar fixes, we select examples from a collection of buggy-fixed pairs based on error message similarity. We call this collection of buggy-fixed pairs the example bank.

To capture differences in language tooling, we implement two methods for selecting programs from our example bank. The key difference between these two methods is how they compute a similarity metric over error diagnostics.

The first variant, *error vector selection*, assumes that fine-

grained error reporting is available. For example, the Excel parser returns a detailed report with many different diagnostic counters. We count the occurrence of each error category reported by the tool and construct a vector out of these frequencies – we refer to this as an error vector. We then select programs from the example bank by minimizing the L2 distance between error vectors.

The second variant, *message embedding selection*, assumes that high-level errors are accompanied by detailed descriptions in natural language. For example, the Python parser often returns the same error (like SyntaxError) for different mistakes and instead exposes additional information through the associated natural language error message. We use this description by embedding the compiler messages with a pretrained CodeBert (Feng et al. 2020) model and comparing embeddings based on cosine similarity.

Figure 4 shows a simplified few-shot prompt with an example, chosen using message embedding, which exhibits the same error (and required fix) as our buggy program. With this prompt, RING's top candidate is the right repair.

```
### Buggy Python
def initial_solution(self, start,
    (max_shares, desired_weight) ):
...
Error: (1) invalid syntax. Error in line
    : 3, span starts 35 and ends: 36.
### Fixed Python
def initial_solution(self, start,
    max_shares, desired_weight ):
...
```

Figure 4: Our *smart selection of few-shots* retrieves relevant buggy-fix examples from an example bank. Shots are retrieved based on a similarity metric over error diagnostics. The shot selected (pink background) displays the same invalid signature-level tuple parameter unpacking (dark red background, **bold**) as our target program. The fixed portion of the shot (green background, **bold**) removes the parentheses.

Candidate Ranking

LLMs achieve variation in their output by iteratively sampling each token from promising candidates. The extent to which less likely tokens can be selected is controlled by a parameter called *temperature*. We can thus generate multiple candidates by controlling the temperature during generation.

The final step in RING is to rank the candidates obtained by querying Codex using the prompt described in the prior two stages. We use a relatively simple (but effective) ranking strategy to order the candidate programs: averaging the log-probabilities of tokens selected during the decoding process and sort the candidates in descending order of their averages.

During development, we found that generating various candidates with higher temperatures – encouraging diverse candidates – and ranking them yields better performance than using lower temperatures such as zero.

Language-Specific Datasets

We evaluate RING on six different languages, ranging from low-code formula languages to popular scripting languages. We describe the dataset, language-specific baseline(s) and evaluation metric for each language.

Excel We use a recently released dataset of 200 Excel repair tasks collected from Excel help forums (Bavishi et al. 2022). Each task consists of an Excel formula with syntax errors, some semantic errors (such as wrong function call arity) and a ground truth repair. We also collect a set of 73 tasks where the Excel formula contains at least one type error and annotated each such formula with a ground truth repair. The final collection consists of 273 Excel repair tasks.

A successful repair exactly matches the ground truth after normalizing tokens like spaces, capitalizing all the identifiers and cell references. We compare RING to the neurosymbolic repair engine LaMirage (Bavishi et al. 2022).

Power Fx Like Excel, we use the recently released 200 Power Fx repair tasks accompanying LaMirage. These tasks consist of syntactic and basic semantic errors, and are collected from help forums and anonymized product telemetry.

We use the same evaluation criteria as in Excel and compare to the neurosymbolic repair engine LaMirage.

Python We evaluate RING on a random sample of 200 syntactically invalid Python code snippets from the dataset used by the SOTA syntax repair tool for Python: BIFI (Yasunaga and Liang 2021). These code snippets were collected from GitHub repositories.

These snippets do not have a ground truth repair. Hence, we employ the same evaluation metric described in the BIFI paper. A repair is successful if the produced program is (1) parsed successfully by the Python 3 parser and (2) has a Levenshtein (Levenshtein et al. 1966) token edit distance less than 5 from the buggy program. The python tokens are generated by the Pygments[3] lexer.

We compare to BIFI, a transformer-based repair system that iteratively trains a *code breaker* that learns to generate realistic errors and a *code fixer* that repairs such errors.

JavaScript We evaluate RING on a random sample of 200 JavaScript (JS) code snippets drawn from the dataset released with TFix (Berabi et al. 2021). Each snippet has at least one error or warning reported by the popular linter ESLint (Tómasdóttir, Aniche, and Van Deursen 2018). In addition to syntax errors, ESLint also reports stylistic issues.

The dataset released by TFix contains a ground truth repair code snippet for each buggy snippet. Both buggy and ground truth code snippets were mined by the TFix authors from GitHub commits. The originally released dataset contains only the part of each code snippet relevant to the error and repair. However, these parts are an arbitrary window around the original fault location. We found that providing these arbitrary windows to Codex resulted in spurious edits, as the snippets had syntax errors that were just an artifact of the windowing. To mitigate this, we extracted the whole function (or whole file, if not in a function) that encompassed the

[3] https://pygments.org/

originally buggy and the repaired code snippets. We refer to these as *extended code snippets*.

We compare our performance to TFix, a fine-tuned T5 (Raffel et al. 2020) model for JS repair. A repair is successful if it matches the ground truth associated with the buggy program. We run TFix on both the original window snippets and on our extended code snippets.

C We evaluate RING on a random sample of 200 C code snippets drawn from the dataset released with DeepFix (Gupta et al. 2017). These programs correspond to real user programs written by students in an introductory programming class and raise at least one compilation error.

We compare to Dr. Repair, a neural repair system that uses graph attention to combine information from the buggy code snippet and the associated compiler message (Yasunaga and Liang 2020). We use their success criterion: a repair must not raise any error messages when compiled using `gcc -w -std=c99 -pedantic`. Following BIFI, a repair must be less than 5 token edits away from the original buggy program.

PowerShell We introduce the novel task of repairing syntax errors in PowerShell commands. To create benchmarks, we searched StackOverflow (StackOverflow) for the word "error" in threads tagged with `powershell`. This resulted in 14,954 threads. We extracted code blocks with least one space from the question and the accepted answer. We keep pairs from question and answer where the question code is invalid and answer code is valid. We judged validity using the PowerShell command `Get-Command -syntax`.

Finally, we manually annotated these candidate tasks from the associated StackOverflow post, confirming each pair was reflective of the original issue and did not have extra changes. When there were changes, we manually simplified the pair or corrected minor issues like new line characters. We kept a final set of 200 task pairs.

There is no existing language-specific engine to compare with, as we introduce this task. A repair is successful if it exactly matches the associated answer code block.

Common Baseline We also use zero-shot Codex as a baseline for all languages. We use the following prompt:

```
Fix bugs in the below code:
### Buggy <language>:
<buggy program>
### Fixed <language>:
```

where `<language>` is replaced with the appropriate language name for the benchmark task. For all the experiments, we used `###` as stop token and `top_p`= 1.0.

Results and Analysis

We first ask: (**RQ1**) how viable is RING's Codex-powered approach for repair across multiple languages? Next, we investigate the extent to which Codex can address each of our conceptual stages. For localization, (**RQ2**) to what extent can RING perform error localization across languages? For code transformation, (**RQ3**) to what extent does our smart selection of few-shots improve performance? Finally, for candidate ranking, (**RQ4**) to what extent can RING rely on Codex's token log probabilities to rank candidates?

RQ1. Viability of Multilingual Repair

Table 1 shows the performance for RING, language-specific repair engines, and a Codex-based zero-shot baseline, across each of our languages. We present the best performing configuration for each language using *top@k* performance metrics (Inala et al. 2022; Poesia et al. 2022; Bavishi et al. 2022), where we consider the top k candidates produced by a system and count the task as solved if any candidate satisfies our correctness criteria.

Smart selection is done via leave-one-out. For languages with ground truth, all other tasks are the example bank for drawing shots. Since the C and Python datasets do not have ground truth pair, we sample an additional 400 programs from their corresponding datasets. We run the best RING configuration (without smart selection) on these 400 programs and pick those that do not raise any diagnostics error. These buggy/correct pairs form the example bank in C and Python.

RING outperforms the state-of-the-art repair engines in top@1 for Excel, Python, and C. For Power Fx, we find that RING's top@3 rate is comparable to the top@1 rate for LaMirage. Furthermore, there is a substantial improvement in RING's top@3 compared to top@1.

In Javascript, we find that TFix applied to the original code snippets obtains a top@1 rate of 0.59 (approximately 7 points higher than that of RING). However, applying TFix to the extended code snippets results in a much lower top@1 rate of 0.09. This performance degradation can be attributed to the substantially longer sequences of the extended code snippets compared to the original code snippets, an average of 208 and 74 T5 tokens, respectively.

In PowerShell (PS), we observe that RING's performance is substantially lower compared to other languages. We hypothesize that this may be a reflection of the (presumed) relative scarcity of PS commands in Codex's training data. Manual inspection of failures also revealed that RING performs fewer edits than required to match the ground truth.

Given this evidence, we conclude that RING's Codex-powered approach can perform multilingual repair. We can contrast this to the substantial effort required to build a language-specific repair engine. TFix, BIFI, and Dr. Repair were trained on 108K, 3M and 1.5M JavaScript, Python, and C code snippets, respectively. LaMirage trained error localizers and rankers based on pointer networks, as well as implemented multiple language-specific rules.

Programs that were not fixed by either RING or the language-specific engines shared some properties. In particular, some of these could be addressed with a combination of iteratively querying Codex and explicit lightweight constraints that enforce language-specific knowledge. For example, we found a Python program that has two issues: an invalid use of the reserved keyword `async` and a missing parenthesis. For the keyword issue, we could query Codex with the buggy program up to the invalid keyword usage, validate that the following token predicted is not a reserved keyword, and then query Codex *again* with the modified buggy code fragment. This is similar to constrained decoding used in Synchromesh (Poesia et al. 2022).

Language	Approach	Top@1	Top@3	Top@50*	Metric	Avg. Tokens
Excel	RING (Abstracted Message, Error Vector)	**0.82**	**0.89**	**0.92**	Exact Match	26 ±14
	LaMirage (Bavishi et al. 2022)	0.71	0.76	-		
	Codex (Chen et al. 2021)	0.60	0.77	0.88		
Power Fx	RING (Compiler Message, Message Embedding)	0.71	0.85	**0.87**	Exact Match	29 ±19
	LaMirage (Bavishi et al. 2022)	**0.85**	**0.88**	-		
	Codex (Chen et al. 2021)	0.47	0.68	0.84		
Javascript	RING (Compiler Message, Error Vector)	0.46	0.59	**0.64**	Exact Match	163 ±106
	TFix (extended code snippets) (Berabi et al. 2021)	0.09	-	-		
	TFix (original dataset) (Berabi et al. 2021)	**0.59**	-	-		
	Codex (Chen et al. 2021)	0.19	0.28	0.39		
Python	RING (Compiler Message, Message Embedding)	**0.94**	**0.97**	0.97	Passes Parser Edit Distance < 5	104 ±150
	BIFI (Yasunaga and Liang 2021)	0.92	0.95	0.96		
	Codex (Chen et al. 2021)	0.87	0.94	**0.98**		
C	RING (Compiler Message, Message Embedding)	**0.63**	**0.69**	**0.70**	Passes Parser Edit Distance < 5	223 ±72
	Dr Repair (Yasunaga and Liang 2020)	0.55	-	-		
	Codex (Chen et al. 2021)	0.40	0.56	0.61		
Powershell	RING (Compiler Message, Message Embedding)	**0.18**	**0.25**	**0.28**	Exact Match	24 ±30
	Codex (Chen et al. 2021)	0.10	0.15	0.18		

Table 1: Comparison of RING with language-specific approaches and a zero-shot baseline that uses Codex. Bold denotes best performance for each language. *For Powershell we compute Top@20, due to rate limiting restrictions. All RING experiments are at 0.7 temperature. RING can outperform language-specific repair engines in Excel, Python, and C. In Javascript, RING is capable of generating the right repair but ranking needs to improve. In Powershell, with no existing baseline, RING performs substantially worse – likely reflective of the lack of Powershell code in Codex's training data. We ran all Codex-related queries on August 9th 2022 using Open AI's public API for "davinci-code-002", with the exception of Powershell experiments which we ran on March 7th 2023.

RQ2. Error Localization

Even if RING cannot fix a program at top@1, locating the error can help users. We carry out the following experiment for the four languages which have the ground truth. We consider programs that are not repaired at top@1 by RING and those that are not repaired at top@1 by the language-specific baseline. For each such program, we take the top candidate produced by each system and compare the edit locations to the ground truth edit locations. If the candidate edit locations are all within a range of ±k tokens of the ground truth locations, we mark this as a correct localization.

Figure 5 summarizes our results. We observe that RING correctly locates a larger fraction of required edits compared to the language-specific baselines. This holds true across the four languages with ground truth repairs. RING's localization success varies by language but can reach as high as over a quarter of unrepaired programs (for Power Fx, given a tolerance of one token). For such programs, where RING can localize the error but does not perform the correct edit, drawing shots from a larger example bank may help.

Next, we explored a key contributing factor to overall repair success (and localization in particular): program length. We found that for most languages, the buggy programs that RING can repair tend to be shorter than the buggy programs it fails to repair. Figure 6 shows the cumulative fraction of

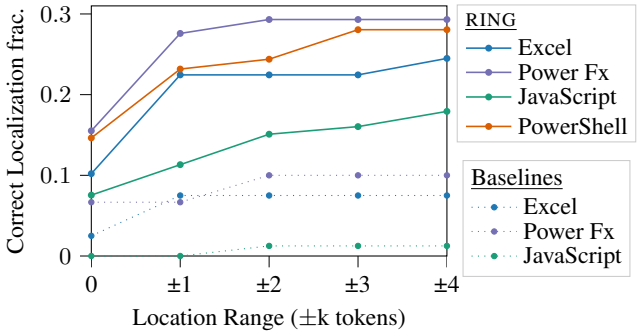
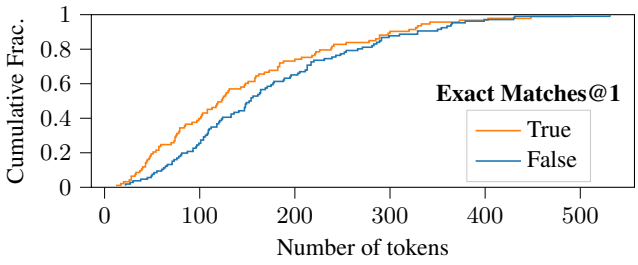
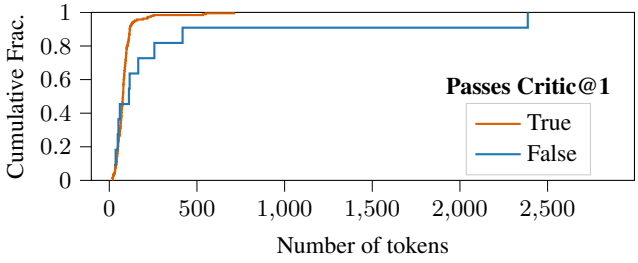


Figure 5: We consider separately the programs not repaired at top@1 by RING and language-specific baselines. We compute an approximate error localization metric, which marks as correctly localized any edit that is within k tokens of the groundtruth edit location. When RING fails to repair a program it correctly localizes a larger fraction of programs compared to the language-specific baselines.

buggy programs by their length, grouped based on their outcome (top@1). In both JavaScript and Python, the programs successfully repaired by RING tend to be shorter than those where it fails. Interestingly, this relationship does not seem

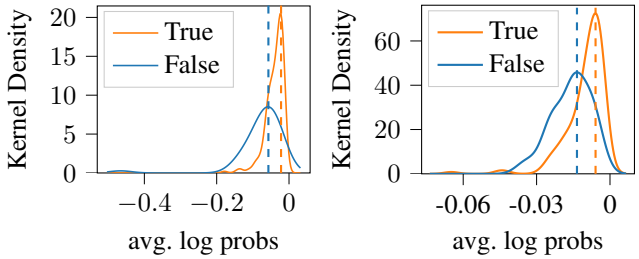


(a) Javascript



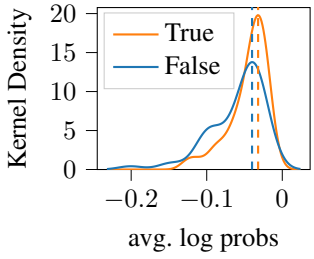
(b) Python

Figure 6: Cumulative fraction of programs by number of tokens in the original buggy program, grouped by whether RING can repair at top@1. Successful repairs tend to be associated with shorter buggy programs.

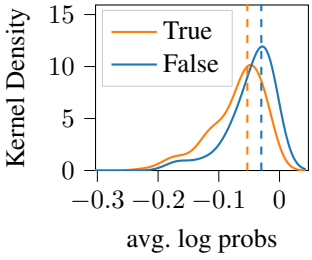


(a) Excel

(b) C



(c) Power Fx



(d) Powershell

Figure 7: (Gaussian) Kernel density plots for average token log probabilities across languages, based on their top@1 success status. Clearer separation of distributions tends to be associated with better performance (e.g., Excel, C). In Powershell, where RING struggles, the relationship between distribution peaks is inverted relative to other languages.

to hold as strongly for Excel. We attribute this behaviour to overall shorter programs lengths and the restrictive Excel grammar.

RQ3. Code Transformation

Table 2 shows the top@1 rate with our smart selection of few-shots for the prompt, compared to a strategy that uses pre-defined fixed examples. The pre-defined strategy allows us to curate high-quality repair examples for common errors, but these may not be relevant for all programs. Prior work (Prenner and Robbes 2021) explored the use of fixed few-shot examples for APR with Codex.

Our results show that smart selection improves performance in all languages. This performance improvement comes from examples in the prompt that reflect similar errors (and expected edits) to the target program. We use error vector selection for Excel and JavaScript, which have better and more granular error categorization, and message embedding selection for other languages. We observe that Power Fx shows the smallest performance improvement. Manual inspection revealed that Power Fx compiler messages tend to be imprecise, and using them to select examples can introduce some noise into the prompt. An example that we encountered were cases where the compiler suggested there was an extraneous token in the input program that did not actually appear in it.

RQ4. Candidate Ranking

RING ranks candidate repairs based on the average of per-token log probabilities produced by Codex. The effectiveness of this strategy for our use case depends on the extent

Language	Fixed Shots	Smart Shots	Fractional Change
Excel	0.76	0.82	0.08
Power Fx	0.70	0.71	0.01
Javascript	0.43	0.46	0.07
Python	0.91	0.94	0.03
C	0.50	0.58	0.16
Powershell	0.15	0.18	0.20

Table 2: Top@1 for few-shots selected using our smart selection strategy, compared to pre-defined fixed examples. Smart selection improves performance for all languages. For Power Fx, we see the smallest improvement, which we attribute to imprecise compiler diagnostics.

to which Codex is calibrated properly for program repair (Bella et al. 2010; Nixon et al. 2019; Dormann 2020). Figure 7 compares average log probabilities in Excel, Power Fx, PowerShell, and C. We show (Gaussian) kernel density plots across languages based on the top@1 outcome. For languages like Excel and C, where RING outperforms language-specific repair engines, there is a clearer difference in distributions. In Power Fx, where RING can repair programs but does not outperform the language-specific engine, this distribution difference is less clear. In PowerShell, where RING fails to repair a substantial fraction of programs, the relationship between the peaks of the distributions is inverted relative to other languages.

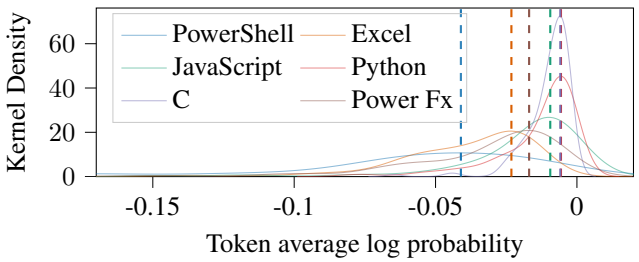


Figure 8: Per-language (Gaussian) kernel density plots of successful top@1 scores (average token log probabilities). We find that less popular languages, like PowerShell, Excel, and Power Fx, have lower average scores – likely reflective of their relatively small fraction of Codex's training data.

Additionally, we find that even for programs with a top@1 success outcome, there are differences in average token log probability across languages, as shown in Figure 8. For less popular languages (PowerShell or Excel), the distribution peaks are further left than more popular languages (JavaScript). This likely reflects the underlying language distribution in Codex's training data.

Based on our observation of the gap between top@1 and top@5, paired with these calibration insights, we believe that a language-specific ranking model (Inala et al. 2022) may provide a substantial payoff in multilingual repair.

Discussion

We now provide discussion on the design principles involved in building a good example bank for few-shot selection and the tasks required to adapt RING to a new language.

Designing the Example Bank

While curating the example bank, it is essential to have different types of errors to facilitate retrieval of similar mistakes/fixes for a new buggy program. There are several ways to collect such examples, including scraping public forums, using telemetry data, and bootstrapping examples through the language knowledge of an expert. We have found that scraping public forums is a good way to start, paired with expert curation of corner cases. Buggy-fixed pairs can be collected incrementally, adding more diverse examples from different sources later. Telemetry data also provides a natural source for examples, but depending on the platform/organization can require anonymization that might impact retrieval.

Our evaluation employs a strict leave-one-out strategy to build an example bank from benchmark programs. In practice, this will be a very restrictive example bank that can potentially limit the number of successful repairs.

While the example bank sizes used during our evaluation do not present a performance concern, as example banks in production grow, retrieval time may become more significant. To address such challenges, RING could take advantage of off-the-shelf fast indexing/retrieval systems, such as FAISS (Johnson, Douze, and Jégou 2019) or ANNOY (Spotify 2022).

Adapting RING for New Languages

We now detail the steps required to apply RING to a new language. The first task is to build the associated example bank, using the principles discussed above. Next, we need to evaluate the language tooling available for error diagnostics. In particular, there are two key decisions: determining what kind of error-based few-shot selection to make and if the error message needs to be abstracted prior to use for localization with RING. We discuss each of these concerns in turn.

Choosing between Error Vector and Message Embedding Different languages have different underlying language tools, such as compilers and linters. If the underlying language tools provide detailed error reports with granular error categories, counting the categories can help us extract precise error information. We recommend using error vector selection for such languages. Unfortunately, not all languages provide fine-grained error categories but instead expose additional information through an associated natural language error message. For such languages, which provide more information through natural language, we recommend using message embedding selection.

Creating abstracted error message When incorporating the error message in the localization portion of the prompt and in message-embedding-based few shot selection, some languages may benefit from abstracting the error message to remove extra (and possibly imprecise) information. In our experiments, we found that providing an error message without exact location information can help in low-code languages like Excel. If the language tool provides data structures with error description, location, and error category, we only use the description. Languages with natural language error messages typically follow a template that we can use to extract the portions of the message we want to preserve to create the abstracted message.

For example, in C, the error message for a missing semicolon (;) at the end of a statement is shown below:

```
In function 'main':
16:6: error: expected ';' before 'printf'
      printf(\"%d\",catalan(h));
      ^
```

We split the error message using regular expression "`\d+:\d+: error:`", which captures the text `16:6 error:` and leaves us with the following abstracted error message: `expected ';' before 'printf'`.

Conclusion

We present RING, a multilingual repair engine powered by Codex. We show various prompt-based strategies designed to convey developer-like information to address the three stages of automate program repair: error localization, code transformation, and candidate ranking. We evaluate RING on six languages, including a benchmark for the novel task of repairing Powershell programs. We show RING can perform well in multiple languages, even outperforming language-specific engines in some, with little engineering effort.

Acknowledgements

We thank Peter Lee for the CoPilot-flipped-model analogy. We thank Julia Liuson for inspiring and facilitating use of Codex-as-a-component in our neuro-symbolic workflows. We also thank the authors of baseline systems used in our research – their sharing of models and data made this work possible. We would also like to thank Abishai Ebenezer for helping us curate the PowerShell evaluation benchmarks.

References

Ahmed, T.; Ledesma, N. R.; and Devanbu, P. 2021. SYNFIX: Automatically Fixing Syntax Errors using Compiler Diagnostics. *arXiv preprint arXiv:2104.14671*.

Ahmed, U. Z.; Kumar, P.; Karkare, A.; Kar, P.; and Gulwani, S. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, 78–87.

Altadmri, A.; and Brown, N. C. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM technical symposium on computer science education*, 522–527.

Arcuri, A. 2008. On the automation of fixing software bugs. In *Companion of the 30th international conference on Software engineering*, 1003–1006.

Bareiß, P.; Souza, B.; d'Amorim, M.; and Pradel, M. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. *arXiv preprint arXiv:2206.01335*.

Bavishi, R.; Joshi, H.; Cambronero, J.; Fariha, A.; Gulwani, S.; Le, V.; Radiček, I.; and Tiwari, A. 2022. Neurosymbolic Repair for Low-Code Formula Languages. *Proc. ACM Program. Lang.*, 6(OOPSLA2).

Bella, A.; Ferri, C.; Hernández-Orallo, J.; and Ramírez-Quintana, M. J. 2010. Calibration of machine learning models. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, 128–146. IGI Global.

Berabi, B.; He, J.; Raychev, V.; and Vechev, M. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, 780–791. PMLR.

Bommasani, R.; Hudson, D. A.; Adeli, E.; Altman, R.; Arora, S.; von Arx, S.; Bernstein, M. S.; Bohg, J.; Bosselut, A.; Brunskill, E.; et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901.

Bureau of Labor Statistics, U. 2022. Software developers, Quality Assurance Analysts, and testers : Occupational outlook handbook. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm. Accessed: 2022-07-30.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Ponde, H.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D. W.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Babuschkin, I.; Balaji, S. A.; Jain, S.; Carr, A.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M. M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. *ArXiv*, abs/2107.03374.

Chowdhury, J. R.; Zhuang, Y.; and Wang, S. 2022. Novelty Controlled Paraphrase Generation with Retrieval Augmented Conditional Prompt Tuning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(10): 10535–10544.

Debroy, V.; and Wong, W. E. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. *2010 Third International Conference on Software Testing, Verification and Validation*, 65–74.

Diekmann, L.; and Tratt, L. 2020. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers. In *34th European Conference on Object-Oriented Programming, ECOOP 2020*, volume 166 of *LIPIcs*, 6:1–6:32.

Dormann, C. F. 2020. Calibration of probability predictions from machine-learning and statistical models. *Global ecology and biogeography*, 29(4): 760–765.

Drori, I.; Zhang, S.; Shuttleworth, R.; Tang, L.; Lu, A.; Ke, E.; Liu, K.; Chen, L.; Tran, S.; Cheng, N.; et al. 2022. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences*, 119(32): e2123433119.

Drosos, I.; Guo, P. J.; and Parnin, C. 2017. HappyFace: Identifying and predicting frustrating obstacles for learning programming at scale. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 171–179. IEEE.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Gazzola, L.; Micucci, D.; and Mariani, L. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering*, 45: 34–67.

Goues, C. L.; Pradel, M.; and Roychoudhury, A. 2019. Automated program repair. *Communications of the ACM*, 62(12): 56–65.

Gupta, R.; Pal, S.; Kanade, A.; and Shevade, S. K. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI*.

Hajipour, H.; Bhattacharyya, A.; and Fritz, M. 2020. SampleFix: Learning to Correct Programs by Efficient Sampling of Diverse Fixes. In *NeurIPS 2020 Workshop on Computer-Assisted Programming*.

Inala, J. P.; Wang, C.; Yang, M.; Codas, A.; Encarnación, M.; Lahiri, S. K.; Musuvathi, M.; and Gao, J. 2022. Fault-Aware Neural Code Rankers. *arXiv preprint arXiv:2206.03865*.

Johnson, J.; Douze, M.; and Jégou, H. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3): 535–547.

Levenshtein, V. I.; et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, 707–710. Soviet Union.

Liu, K.; Li, L.; Koyuncu, A.; Kim, D.; Liu, Z.; Klein, J.; and Bissyandé, T. F. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171: 110817.

Murphy, L.; Lewandowski, G.; McCauley, R.; Simon, B.; Thomas, L.; and Zander, C. 2008. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin*, 40(1): 163–167.

Nguyen, H. D. T.; Qi, D.; Roychoudhury, A.; and Chandra, S. 2013. SemFix: Program repair via semantic analysis. *International Conference on Software Engineering*, 772–781.

Nixon, J.; Dusenberry, M. W.; Zhang, L.; Jerfel, G.; and Tran, D. 2019. Measuring Calibration in Deep Learning. In *CVPR Workshops*, volume 2.

Parihar, S.; Dadachanji, Z.; Singh, P. K.; Das, R.; Karkare, A.; and Bhattacharya, A. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, 92–97.

Poesia, G.; Polozov, A.; Le, V.; Tiwari, A.; Soares, G.; Meek, C.; and Gulwani, S. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *International Conference on Learning Representations*.

Prenner, J. A.; and Robbes, R. 2021. Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs. *arXiv preprint arXiv:2111.03922*.

Pu, Y.; Narasimhan, K.; Solar-Lezama, A.; and Barzilay, R. 2016. sk_p: a neural program corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 39–40.

Qi, Y.; Mao, X.; Lei, Y.; Dai, Z.; and Wang, C. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, 254–265.

Radford, A.; Narasimhan, K.; Salimans, T.; Sutskever, I.; et al. 2018. Improving language understanding by generative pre-training. https://paperswithcode.com/paper/improving-language-understanding-by. Accessed: 2022-08-05.

Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; and Liu, P. J. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140): 1–67.

Shannon, C. E. 1948. A mathematical theory of communication. *The Bell system technical journal*, 27(3): 379–423.

Spotify. 2022. ANNOY library. https://github.com/spotify/annoy. Accessed: 2022-08-01.

StackOverflow. 2022. StackOverflow Website. https://stackoverflow.com/.

Tómasdóttir, K. F.; Aniche, M.; and Van Deursen, A. 2018. The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering*, 46(8): 863–891.

Wexelblat, R. L. 1976. Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. In *Proceedings of the 2nd international conference on Software engineering*, 331–336.

Yasunaga, M.; and Liang, P. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, 10799–10808. PMLR.

Yasunaga, M.; and Liang, P. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*, 11941–11952. PMLR.

Zhong, H.; and Su, Z. 2015. An Empirical Study on Real Bug Fixes. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 1: 913–923.