Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example

MALINDA DILHARA, University of Colorado, USA
ABHIRAM BELLUR, University of Colorado, USA
TIMOFEY BRYKSIN, JetBrains Research, Cyprus
DANNY DIG, JetBrains Research, University of Colorado, USA

Software developers often repeat the same code changes within a project or across different projects. These repetitive changes are known as "code change patterns" (*CPATs*). Automating *CPATs* is crucial to expedite the software development process. While current Transformation by Example (TBE) techniques can automate *CPATs*, they are limited by the quality and quantity of the provided input examples. Thus, they miss transforming code variations that do not have the exact syntax, data-, or control-flow of the provided input examples, despite being semantically similar. Large Language Models (LLMs), pre-trained on extensive source code datasets, offer a potential solution. Harnessing the capability of LLMs to generate semantically equivalent, yet previously unseen variants of the original *CPAT* could significantly increase the effectiveness of TBE systems.

In this paper, we first discover best practices for harnessing LLMs to generate code variants that meet three criteria: correctness (semantic equivalence to the original *CPAT*), usefulness (reflecting what developers typically write), and applicability (aligning with the primary intent of the original *CPAT*). We then implement these practices in our tool PyCraft, which synergistically combines static code analysis, dynamic analysis, and LLM capabilities. By employing chain-of-thought reasoning, PyCraft generates variations of input examples *and* comprehensive test cases that identify correct variations with an F-measure of 96.6%. Our algorithm uses feedback iteration to expand the original input examples by an average factor of 58x. Using these richly generated examples, we inferred transformation rules and then automated these changes, resulting in an increase of up to 39x, with an average increase of 14x in target codes compared to a previous state-of-the-art tool that relies solely on static analysis. We submitted patches generated by PyCraft to a range of projects, notably esteemed ones like *microsoft/DeepSpeed* and *IBM/inFairness*. Their developers accepted and merged 83% the 86 CPAT instances submitted through 44 pull requests. This confirms the usefulness of these changes.

CCS Concepts: • **Software and its engineering → Programming by example**; **Software maintenance tools**; • **Computing methodologies → Artificial intelligence**.

Additional Key Words and Phrases: Transformation by Example, Program by Example, Python, Code Changes, Automation, Test Case Generation, Large Language Models, Generative AI, Machine Learning, Code Clone

ACM Reference Format:
Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example. *Proc. ACM Softw. Eng.* 1, FSE, Article 29 (July 2024), 23 pages. https://doi.org/10.1145/3643755

Authors' addresses: Malinda Dilhara, University of Colorado, Boulder, USA, malinda.malwala@colorado.edu; Abhiram Bellur, University of Colorado, Boulder, USA, Abhiram.Bellur@colorado.edu; Timofey Bryksin, JetBrains Research, Limassol, Cyprus, timofey.bryksin@jetbrains.com; Danny Dig, JetBrains Research, University of Colorado, Boulder, USA, danny.dig@colorado.edu.

ACM 2994-970X/2024/7-ART29
https://doi.org/10.1145/3643755

1 INTRODUCTION

Software developers frequently change code to improve performance, manage resources efficiently, integrate new libraries, etc. Throughout this process, they frequently repeat identical or similar code modifications [7, 17, 30, 40, 41]. These repetitions stem from the adoption of shared coding idioms [3, 12, 44, 67], adherence to common best practices [16, 34], and the need to tackle similar programming challenges [33, 55]. Such repeated changes occur at a fine-grained level, frequently appearing within specific methods, and consistently retaining the same semantics.

For example, Listing 1 shows such a repeated change in project *NifTK/NiftyNet*, an open-source convolutional neural network platform. The developers replaced a `for` loop that summed `elements` of a list with the more efficient `numpy.sum` function, which is a best practice and improves performance. The performance gain is attributed to several factors, including NumPy's C implementation, vectorization, memory efficiency, optimized algorithms, and parallel processing support. This change involves specific programming idioms and is localized to a particular method within the project. This change recurs at multiple locations across various commits, prompting previous researchers [14, 16, 41] to identify these recurrent changes as *code change **patterns*** (*CPATs*).

Developers often perform these changes manually [16, 41]: they first must identify potential target code sites and then apply the required syntax transformations. This manual process is time-consuming, tedious, and error-prone due to two main reasons: (i) identifying all potential target sites to apply *CPATs* is challenging, as they may be deeply embedded within the code, exhibiting syntactic variations along with variations in data- and control-flow, and (ii) ensuring consistency and correctness in applying identical changes across multiple locations in the codebase is challenging, as developers must reason about syntactic variations. To address these challenges and enhance developer productivity, researchers have employed "Transformation by Example" (TBE) [5, 13, 20, 22, 26–29, 31, 33, 35, 35, 37, 39, 57, 58, 65] techniques to automate code changes. These techniques infer transformation rules from before-and-after versions of code changes and use these inferred rules to automatically transform new target code sites that exhibit syntactical similarities and have a program structure similar to the original code change. These approaches are effective in API migrations, e.g., replacing obsolete API calls with modern ones from the Android SDK [20, 21, 25, 26, 35, 65], type migrations in Java [33], and API migrations in Linux systems [55].

Despite their successes, existing TBE techniques face challenges when handling more complex coding idioms, where there can be numerous potential target sites that are semantically equivalent, yet differ in terms of syntax and data- and control-flow. These techniques are limited to transforming target codes that *perfectly resemble* the input example code change and struggle with variations beyond those specific examples. We call these *previously unseen variants*. Unseen variants can be of two types: (i) syntax variants, and/or (ii) data- or control-flow variants. Listing 2 shows an example of syntax variant where the code computes the sum of elements in the list `losses` by using a different syntax involving `len` and `range`. It is also a data-flow variant through list indexing and accumulation in the variable `loss`. Notably, this target code would not be identified or transformed by existing techniques that inferred transformation rules based on Listing 1 as input. Our initial analysis uncovered 50 other ways to compute the sum of elements, all of which are unseen variants of Listing 1. They could all benefit from and be sped up by transforming them to `np.sum`. Sadly, existing techniques fail to identify and transform these unseen variants.

Listing 1. Commit c8b28432 in GitHub project NifTK/NiftyNet: Replace `for` loop with `numpy.sum`

```
-result = 0
-for elem in elements:
-    result = elem + result
+result = numpy.sum(elements)
```

Listing 2. GitHub repository CDE-GAN/models employs a `for` loop for calculating the sum of an array similar to the tranformed code in Listing 1

```
loss = 0
for i in range(len(losses)):
    loss += losses[i]
```

Recent advancements, *PyEvolve* [14] and *Spinfer*[55], are effective in handling certain unseen data- or control-flow variants. However, none of the existing tools are equipped to handle syntax variants. While *Spinfer* [55] addresses control variations through the use of the "..." operator to represent arbitrary statements between specific statements in input examples, it relies on multiple input examples to learn the potential locations for inserting "..." in the rule. This reliance on examples can be problematic if not all potential locations for arbitrary statements are covered, limiting its effectiveness in handling many unseen variants. On the other hand, *PyEvolve* [14] excels in automating unseen variants by employing a graph-based approach to support variations not exemplified in the input examples, specifically those concerning data- and control-flow. However, this approach is limited to automating simple data and control variations, such as reassigning values to other variables, and it cannot handle more complex variations, as seen in Listing 2, where list elements are accessed using indexing (`loss += losses [i]` in Listing 2).

To advance the frontier for "Transformation by Example" systems, we designed, implemented, and evaluated a novel approach and a tool, *PyCraft*. It successfully automates unseen variants, even those with *completely different syntax*. *PyCraft* harnesses the creativity of Large Language Models (LLMs) to generate many syntactical variations for a given code idiom. LLMs are robust machine learning models that are trained on vast datasets, which encompass source codes as well as documents related to software development. Models like GPT-3.5 [9] and GPT-4 [42] generate coherent and contextually relevant code snippets in response to given prompts. Researchers demonstrated LLMs' versatility across various software engineering tasks, including code completion [11], refactoring [46, 47], code summarization [18, 23], and bugs reproduction [56].

In *PyCraft*, we employ few-shot learning to generate unseen variants for the before-part of a given input *CPAT*. Our initial analysis found that 65.5% of the LLM-generated variants are either erroneous or not semantically equivalent to the original *CPAT* and cannot be used. Therefore, we discovered three criteria that the generated variants must meet: (i) semantically similar to the original input (i.e., correct), (ii) practicality in terms of what developers typically write (i.e., useful), and (iii) aligned with the structural intent of the original *CPAT* (i.e., applicable). To achieve this, we carefully fine-tune the hyper-parameters of the LLM, and conduct comprehensive automatic checks, including static code validation, to ensure correctness and verify that the variants adhere to the original *CPAT*'s structural intent. We also perform dynamic analysis through automatically generated test cases to ensure conformity with the desired behavior. To optimize LLM performance, we fine-tune the hyper-parameters by discovering best practices for generating variants and test cases, finding that higher temperature values are effective for dynamic analysis-focused test case generation, while intermediate temperatures help reduce non-useful variants. This thorough process guarantees the reliability and effectiveness of *PyCraft*. Finally, *PyCraft* infers transformation rules that are used to apply *CPAT* to new target codes including unseen variants, significantly enhancing state-of-the-art TBE techniques and further advancing their automation capabilities.

We conducted a comprehensive evaluation of *PyCraft* to assess its effectiveness in generating variants and the usefulness of the resulting code changes. We utilized *CPATs* mined from open-source repositories as input for *PyCraft* and observed that it could generate raw variations of up to 584 per *CPAT*, with an average *459* per *CPAT*. To obtain high-quality applicable variants, we use a combination of static and dynamic analysis to validate the raw variants. *PyCraft* consistently generated applicable variants at an average *58* per *CPAT* while effectively eliminating inapplicable variants, on average *748* per *CPAT*. This not only demonstrates its proficiency in identifying irrelevant variants but also its ability to successfully infer previously unattainable rules using state-of-the-art techniques. Furthermore, to quantitatively compare *PyCraft* and the previous state-of-the-art, we conducted a comparison analysis using *PyCraft* and *PyEvolve*, the leading tool for automating unseen variations. *PyCraft* exceeded the baseline by enabling an average of 14x

Listing 3. Tranformation rule for the code change in Listing 1

```
:[[v0]] = 0                                    ⇒    :[[v0]]=numpy.sum(:[[v2]])
for :[[v1]] in :[[v2]]:
    :[[v0]] = :[[v0]] + :[[v1]]
```

Listing 4. The repository Yolov uses a `for` loop to compute the sum of an array

```
temp_list = [0] + int_list
for i in range(1, len(temp_list)):
    temp_list[i] += temp_list[i - 1]
count = temp_list[-1]
```

additional code transformation instances per *CPAT*; these would have been missed by prior tools. Furthermore, to assess the usefulness of the generated variations, we submitted pull requests to highly-rated projects including *microsoft/DeepSpeed* and *IBM/inFairness*, totaling 86 CPAT instances. At the time of this writing, developers have accepted 72 (83%) of the CPAT instances, submitted through 44 pull requests. This confirms the practical value of the transformations performed by *PyCraft* for the real-world developers.

This paper makes the following key contributions:

(1) We pioneer a new approach that utilizes LLMs in TBE to generate unseen variants, thereby automating code changes that were once unattainable with existing TBE techniques.
(2) We provide best practices for using LLMs to generate code variations and their test cases.
(3) We designed, implemented, and evaluated these ideas in a new tool, *PyCraft*. We perform comprehensive experiments, including a performance evaluation of *PyCraft* and a detailed qualitative analysis, to demonstrate the capabilities of LLMs in generating variations and the effectiveness of our technique for selecting the applicable ones for automation. Moreover, we conduct a user study to further validate the usefulness of our approach.
(4) Our tool and evaluation dataset is open-source and available for others to reuse [49].

2 MOTIVATING EXAMPLES

To demonstrate the challenges faced by current techniques that use "Transformation by Example" on *CPATs*, we employ real-world code changes. First we show a scenario when existing techniques exhibit a high recall rate, effectively automating such changes. Then, we present two complex scenarios when existing techniques fail to automate code changes, whereas *PyCraft* is successful.

Listing 1 shows an example of a *CPAT* mined from the project *NifTK/NiftyNet*. The developer transforms the `for` loop that computes the sum to `np.sum()`, a highly optimized API from the library *NumPy*. The transformation is represented by the rewrite rule shown in Listing 3. The rule, following ComBy syntax [60], has a left hand side (LHS – indicating the "before" change) and a right side (indicating the "after" change) separated by an arrow. Both sides of the rule contain Python statements with template variables (e.g., `:[[v0]]`) that bind to AST nodes in the actual source code (e.g., `:[[v0]]` binds to `result`). The right side represents the code fragments that replace the left side. These rules can be applied to transform any target code that shares the same AST structure as the code presented in Listing 1, irrespective of any variations in variable names. Notably, existing TBE techniques infer this rule and automate the transformation of structurally similar target codes.

However, many real-world *CPATs* involve semantically equivalent different variations [14, 16]. These variations can be identified in two different ways. The first type, Variant Type 1 (*VT1*), consists of code fragments that possess different syntax but exhibit similar semantics, resembling Type 4 clones [53]. The second type, Variant Type 2 (*VT2*), comprises code fragments that are syntactically equivalent but differ in terms of data- and control-flow.

Listing 2 is an example of *VT1*. It iterates over a list to compute the sum of elements in the array `int_list` similar to Listing 1. However, unlike Listing 1, it utilizes the `len()` function to determine the length of the list `int_list`. This length value is then used with the `range()` function to generate a sequence of numbers. The loop iterates over this sequence, accessing each indexed

Listing 5. An unseen variation generated by LLM for the LHS in Listing 1

```
result = 0
for i in sorted(elements):
    result += i
```

Listing 6. Transformation rule that need to be inferred to transform code in Listing 2

```
:[[v0]] = 0
for :[[v1]] in range(len(:[[v2]])):      ⇒    :[[v0]]=numpy.sum(:[[v2]])
    :[[v0]]  += :[[v2]][:[[v1]]]
```

element of `int_list` for performing the sum operation. This code fragment comes from project CDE-GAN/models and is semantically similar to the Listing 1. However, it differs syntactically, making it a Type-4 clone of the Listing 1. Notably, this target code should be similarly transformed into the `np.sum()` function, as demonstrated in the *CPAT* given in Listing 1. The rule shown in Listing 6 is essential for transforming the code presented in Listing 2. Since Listing 2 was not seen as a change exemplar, existing TBE techniques are unable to infer the rule in Listing 6, resulting in their failure to transform this new target site. In contrast, *PyCraft* succeeds in this task.

Listing 4 represents a *VT2* variant that computes the sum of elements in the `int_list` array, similar to Listing 1. It differs from Listing 1 by assigning the initial list to the temporary list `temp_list` and computing the cumulative sum of the array elements, rather than using an accumulator variable. The last element in the list is the sum of all the elements. This variation in data-flow classifies it as a data-flow variant, thereby denoting it as a *VT2* variant. Among the existing TBE tools, *PyEvolve* [14] can partially automate these *VT2* variants. It captures control-flow variations that include unrelated nodes among the pattern's statements, as well as data-flow variations such as reassigning a variable to another variable that is not exemplified in the *CPAT*. However, it falls short in automating the transformation of unseen variations that contain complex data-flow relations, such as those observed in Listing 4: (i) reassigning the cumulative sum to the array elements, and (ii) differentiating between the use of the "addition assignment" operator (`temp_list[i] += temp_list[i-1]`) and the "addition and assignment" operators (`result = elem + result`). Therefore, existing techniques can not fully automate complex *VT2* variations; whereas *PyCraft* succeeds.

The key idea behind *PyCraft* is to generate *VT1* and *VT2* unseen code variations using an LLM. However, not all of the generated variants can be directly used for rule inference. Some may contain syntax errors, import issues, incorrect types, or lack semantic equivalence. Our empirical evaluations show that these erroneous variants can account for as much as 76% of the total generated variants. Even if 24% of them are correct, not all of these correct variants are suitable for the final rule inference, as certain semantically equivalent correct variants may not be useful. For example, the variant in Listing 5 performs an unnecessary sorting operation that an actual developer would not typically perform, making it a *not-useful* variant. Our empirical evaluations indicate that these not-useful variants can constitute up to 80% of the total correct variants and significantly impact the tool's performance. Therefore, it is essential to reduce them while increasing the useful variants.

Even when certain variants are indeed useful, some may still deviate from the original structural intent. For example, the variant `sum(elements)`, generated by LLM to calculate the sum of elements in an array, is semantically equivalent to Listing 1 and useful. However, it does not iterate a collection as in Listing 1, deviating from the intended structural intent, making it not applicable. We observed that these non-applicable variants can account for up to 70% of the useful variants. Thus, *PyCraft* selects correct, useful, and applicable variants (e.g., Listing 2) for the final rule inference.

3 TECHNIQUE

In this section, we present how our tool, *PyCraft*, automates unseen variations of *CPATs*. Figure 1 shows the overall architecture of *PyCraft*. First, to extract *CPATs*, we use *R-CPATminer* [16], which has been shown to be highly effective for extracting *CPATs* from the version history of open-source repositories. Each *CPAT* includes multiple code transformations extracted from real-world

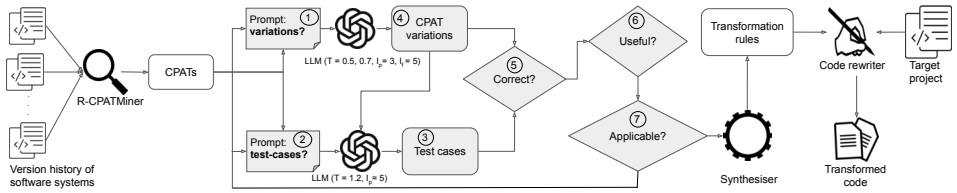

Fig. 1. Schematic diagram of *PyCraft*

repositories. Then, *PyCraft* takes *CPATs* as input and invokes an LLM (Step ① in Figure 1) to generate unseen variations for original coding idioms involved in the *CPAT*. To ensure correctness, *PyCraft* validates the generated code variations by checking the syntax errors and semantic equivalence against the original *CPAT* (Step ⑤ – detailed in Section 3.1.3). Moreover, through the additional fine-tuning parameters of *PyCraft* (Step ⑥) we generate more useful variations and reduce those that are not useful (detailed in Section 3.1.4). This step is crucial in identifying realistic variants that are likely to be present in actual code-bases. The final step involves static code analysis to identify applicable variations (Step ⑦) that align with the original *CPAT*'s intent (detailed in Section 3.1.5). Next, *PyCraft* synthesizes a set of transformation rules using both human adaptations mined by *R-CPATminer* and the generated variants. Finally, *PyCraft* applies the synthesized transformation rules, generating edits for other target codes. *PyCraft* leverages the core components of *PyEvolve* to infer and apply rules, while *PyEvolve* achieves transformations with precision at *97%* and a recall of 94%, ultimately aiding *PyCraft* in reliably executing transformations. These edit suggestions are then presented to developers, who can decide whether to apply them. We will present the technical details of each step in the following.

Definition 3.1. (Unseen variant) Let C be a CPAT consisting of code change instances: $C = \{c_1^{LHS} \rightarrow c^{RHS}, c_2^{LHS} \rightarrow c^{RHS}, \ldots\}$. An unseen variant is defined as $v_u^{LHS} \rightarrow v^{RHS}$, where each $c_i^{LHS} \in C$ exhibits either syntactic differences ($\Delta_{\text{syntax}}(v_u^{LHS}, c_i^{LHS}) \neq \{\}$) or alterations in data or control flow ($\Delta_{\text{data/control}}(v_u^{LHS}, c_i^{LHS}) \neq \{\}$), or both, with respect to v_u^{LHS}. The unseen variant (v_u^{LHS}) preserves the semantic similarity with the original CPAT.

Definition 3.2. (Correct variant - $\mathcal{V}_c$) Let $\mathcal{V}$ be a set of unseen variants generated by an LLM for the CPAT C. A correct variation, denoted as $\mathcal{V}_c$, is a subset of $\mathcal{V}$ that consists of variations semantically equivalent to the original input c_1^{LHS}. Formally, we can express this as: $\mathcal{V}_c = \{v \in \mathcal{V} \mid S(v, C) \text{ holds}\}$, where $S(v, C)$ denotes the semantic equivalence between a variation v and the original CPAT C.

Definition 3.3. (Useful variant - $\mathcal{V}_u$) is a subset of $\mathcal{V}_c$, denoted as $\mathcal{V}_u$, that represents variations that actual programmers would realistically write in their codes. Formally, we can express this as: $\mathcal{V}_u = \{v \in \mathcal{V}_c \mid P(v)\}$, where $P(v)$ represents the condition that a variation v is considered realistic and likely to be written by programmers. The term "not-useful" is used to describe the correct variants that do not belong to the set of useful variants ($\mathcal{V}_c \setminus \mathcal{V}_u$).

Definition 3.4. (Applicable variant - $\mathcal{V}_a$) is a subset of $\mathcal{V}_u$, denoted as $\mathcal{V}_a$, that represents variations with the same structural intent as the original *CPAT*. Formally, this can be expressed as: $\mathcal{V}_a = \{v \in \mathcal{V}_u \mid A(v)\}$, where $A(v)$ represents the condition that a variation v is considered to share the same structural change as the original *CPAT*.

3.1 Generating Unseen Variations

The first step of our approach is to generate a comprehensive set of unseen variations for LHS of each *CPAT* that belongs to the two types, Variation Type 1: *VT1* and Variation Type 2: *VT2*. Here,

we consider the LHS (*i.e.*, before code) of the *CPAT* as the focus for generating these variations since the *CPAT* specifies that the LHS must be transformed into the RHS (*i.e.*, the after code).

3.1.1 Prepare LLM. LLMs initially trained on extensive datasets may require additional preparation for specific tasks, utilizing in-context learning [63]. It provides the LLM with a prompt that prepares it for a particular prediction task. As LLMs advance, in-context learning has evolved into zero-shot learning, enabling predictions solely based on desired output. However, its application to unexplored tasks remains challenging [9, 19, 50]. To overcome this challenge, we employ few-shot learning, augmenting the context with a small number of examples representing desired inputs and outputs. In our work, given that LLMs lack explicit training in understanding, analyzing, and rewriting variants, we utilize few-shot learning [24] as the in-context paradigm to facilitate LLMs in rewriting variants of *CPATs*. We follow best practices as discussed by Gao et al. [24] and include two examples for *VT1* and *VT2*. In addition to examples, the prompt provides LLM with formatting instructions for consistent integration with the tool.

3.1.2 Optimize Tuning Parameters and Generate Variations. The output of an LLM depends on the internal variable "Temperature" (T), that serves as a regulator for the model's output randomness. Higher values, such as 0.9 or 1.2, produce more diverse and unpredictable outputs, whereas values closer to 0 produce focused and less diverse results. Adjusting T allows us to balance between creativity and determinism of the output. Furthermore, the output can exhibit variability when the LLM is presented with the same prompt multiple times. This because: (i) LLMs generate responses using a combination of learned patterns and random sampling, introducing slight variations in answers each time due to the inherent randomness in the generation process; and (ii) LLMs can explore diverse solutions and improve their responses iteratively through feedback [9, 50], enhancing their outputs in subsequent iterations. We primarily focus on two types of iterations in this work: (i) prompting the LLM with the exactly same prompt that consists of the same variant (v_i), known as prompt iteration (I_p), and (ii) changing the prompt to augment it with another semantically similar applicable variation generated in a previous step (v_j where $j! = i$), referred to as feedback iteration. (I_f). While more iterations yield more variations, it also increases processing time and may produce many non-useful variants. Therefore, selecting the ideal combination of temperature and iteration values is of paramount importance. Our goal is to fine-tune the parameters T, I_p, and I_f to maximize the potential for generating diverse and useful variations using the LLMs. Section 4.3 explains our empirical approach to choosing these values.

Not all generated variants are valid for automation. The variations must be correct (Definition 3.2), useful (Definition 3.3), and applicable (Definition 3.4). In the following sections, we explain how we validate the generated *CPATs* to ensure they satisfy these requirements.

3.1.3 Selecting Correct Variations. LLM produces many unseen variations for *CPATs* during the initial step described in Section 3.1. Among these variations, it is crucial to identify the correct ones (Definition 3.2) that are syntactically correct and semantically equivalent to the original *CPAT*, thereby avoiding any erroneous code edit suggestions to the developers.

To ensure the correctness of variations produced by LLMs, we employ a four-step validation process: (i) Syntax validation, (ii) Type validation, (iii) Import validation, and (iv) Semantic validation. Syntax validation checks whether the generated variant forms a valid program, devoid of syntax errors such as indentation errors, parentheses errors. Type validation ensures that variables in both the *CPAT* and the variant maintain the same type. For example, the corresponding variable in the generated variant for the variable `elements` from Listing 1 must be of type *List[int]*. The import validation step ensures the presence and proper usage of required libraries, modules, or dependencies, verifying that APIs and classes in generated variations originate from the same

sources as those used in the original *CPAT*. As the variations are partial code fragments, static code analysis alone cannot infer type information or required imports. To address this issue, we utilize LLMs for type and import inference via chain-of-thought reasoning [63], which leverages rationales as intermediate steps for LLMs to infer required types and imports. This approach facilitates efficient inference of types and imports to evaluate variant conformity with the original *CPATs*.

The final step, semantic validation, assesses semantic equivalence between generated variations and the original *CPAT*. To achieve this, we depend on test cases. However, the original *CPATs* lack pre-existing test cases, necessitating the generation of new ones. Automating test case generation is essential for a streamlined pipeline. We achieve this by prompting LLMs to generate test cases for the original *CPAT*. These test cases validate the semantic equivalence of the generated variations.

To select valid test cases, we follow three steps: (i) We check that the generated test cases are free of syntax errors, (ii) We run the test against the original *CPAT* and disregard the tests that fail. This can happen because the generated tests might have invalid assertions, and (iii) we check that each test initializes every input variable to the *CPAT*. This last step removes tests that do not instantiate all its variables. Such tests may pass the test-case however, using such a test would weed out some correct unseen variants further down the pipeline. In Section 4.2, we quantitatively study the significance of each of these steps, while in Section 4.4, we analyse the quality of the test cases, ensuring their effectiveness in assessing the semantic equivalence of the generated variations.

3.1.4 Selecting Useful Variations. Useful variations (Definition 3.3) capture common coding patterns, idioms, or practices that are typically observed in real-world code. Minimizing the generation of non-useful variants is crucial to limit the inclusion of rules that might not be applicable to the target code. This is because, when applying the rules to codebases, the code rewriter iterates through all the rules to find matching target codes. If there are too many rules that are unlikely to find opportunities for application, it can reduce performance. However, it is not necessary to completely remove non-useful variations, as they are correct transformations that are semantically equivalent to the original *CPAT*. Furthermore, it is impossible to completely remove them. Therefore, it is acceptable to keep them as variations even if they may not find suitable opportunities for application. Our goal is to minimize the number of non-useful examples generated by LLMs while increasing the useful ones. To achieve this, we carefully select parameter values for T and I as explained in Section 4.3, by empirically studying its output for varying parameters.

3.1.5 Selecting Applicable Variations. Applicable variations, a subset of useful variations, align with the structural intent of the original *CPAT*. Filtering out inapplicable once is crucial to prevent unexpected transformations and ensure that only applicable variations, which preserve desired semantics and adhere to intended transformations, are applied to the code bases.

To identify applicable variants, we assess structural intent using three rules:

(i) Control Nodes, as defined by Nguyen et al. [41], are a generalization of AST nodes used to construct fine-grained program-dependence graphs that can be used to group similar code fragments. Control nodes are a group of control statements, such as `if`, `for`, `while` statements. We first check whether all the control nodes ($ControlNodes(C^{LHS}_{AST})$) in the LHS of the original *CPAT* are present in the AST of the generated variant (V_{AST}). Formally, $\forall n \in ControlNodes(C^{LHS}_{AST}) : n \in V_{AST}$.
(ii) V_{AST} does not contain new declarations, such as method declarations (except variable declarations) that do not exist in the AST of CPAT (C^{LHS}_{AST}).
(iii) The sign of difference between the AST nodes in the variant and the RHS of the *CPAT* ($V_{AST} - C^{RHS}_{AST}$) should match the sign of difference of the number of AST nodes between the LHS and the RHS of the *CPAT* ($C^{LHS}_{AST} - C^{RHS}_{AST}$). Formally, $(V_{AST} - C^{RHS}_{AST}) \times (C^{LHS}_{AST} - C^{RHS}_{AST}) > 0$.

In the first rule, comparing control nodes between the original *CPAT*'s LHS and the generated variant is crucial for determining whether the variant captures the essential control-flow structure intended by the original *CPAT*. The second rule ensures that the variant does not contain unexpected statement declarations, such as new methods or even classes. In the final rule, by comparing differences in AST node counts, it identifies whether the variant maintains the desired structural transformation, with consistency in signs indicating that the variant introduces or removes the expected AST nodes as *CPAT*. These rules play a crucial role in selecting applicable variants as they filter out variants that useful but do not adhere to the desired structural intention of the *CPAT*.

3.2 Synthesising Transformation Rules

TBE techniques infer transformation rules that facilitate code change automation, as exemplified in the code change used to infer the rule. A complete transformation rule comprises two components: the *Rule* and the *Guard* [5, 14, 25]. The *Rule* defines specific changes to code fragments, while the *Guard* determines which code the rule should be applied to based on various validations. For example, in Listing 3, the guard for `:[[v2]]` is `type : List[int]`, which verifies whether the corresponding program element in the target code is a list of integers. There is one or many such validations in a Guard that should be considered when deciding where to apply a rule.

We utilize PyEvolve [14] as our rule synthesizer, which leverages InferRule [33] to infer rules. To create a complete code change, we combine each LLM-generated variant with the RHS of the original CPAT and send it to the rule synthesizer. The rule synthesizer infers a rule for the input code change. For example, *PyCraft* inferred the rule given in Listing 6 by taking the input code change in Listing 1, enabling the transformation of the unseen variant shown in Listing 2.

PyEvolve relies on the output of R-CPATMiner [16] to infer guards for the *CPATs* extracted from the version history of repositories. To infer guards for the program elements in LLM-generated variations, *PyCraft* follows a two-step process: (i) If an element e is in both the original *CPAT* (C_{orig}) and a variant (V_a), then the guard validations for e are inferred from output of R-CPATMiner, (ii) if an element e is in a variant (V_i) but not in the original *CPAT* (C_{orig}), then the guard validations for e generated from LLMs. Integrating the rule and guard, *PyCraft* infers a comprehensive transformation rule, ensuring systematic and effective code changes.

4 EVALUATION

We empirically evaluate *PyCraft* by answering the following research questions:

RQ1. How effective are LLMs at generating variations? We depend on LLMs to produce unseen variants, but the ability of LLMs to generate these variants is unknown. Therefore, we perform a quantitative analysis of the variants generated by the LLMs with 20 *CPATs* and utilize the three most recent and largest known LLM models to date.
RQ2. How effective are LLMs at generating test-cases? While we rely on LLMs to generate test cases for dynamic analysis of the variants, their ability for this task is unknown. Therefore, we quantitatively assess the test cases generated by these LLMs across 20 *CPATs*
RQ3. *What are the optimal parameters for generating unseen variants?* The randomness and exploratory ability of LLM vary with temperature (T) and iteration values (I). Studying how LLMs change their output along with T, I values is crucial for enabling *PyCraft* to harness the full potential of LLMs. To achieve this, we study how quality and quantity of variant generation varies with the Temperature and Iterations values.
RQ4. What are the optimal parameters for generating test cases? Tool employs a three-step validation process (Section 3.1.3), to select valid test cases. To fully harness the potential of LLMs,

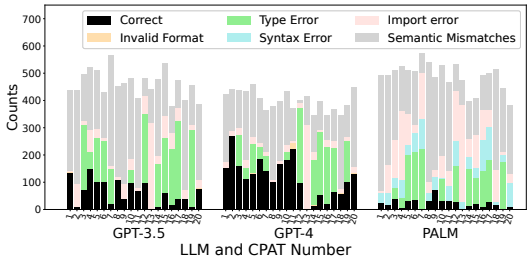

Fig. 2. Variation generating capabilities of each LLM

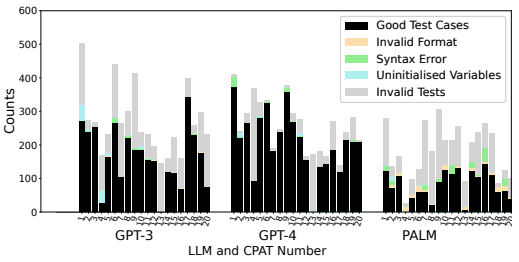

Fig. 3. Unit-test generating capabilities of each LLM

it is crucial to study how the effectiveness (i.e., quality and quantity) of the test cases varies with the parameters Temperature and Iterations.

RQ5. How effective is *PyCraft* at finding new opportunities and performing transformations over the baseline? We want to evaluate the improvements over the previous state-of-the-art tools that are solely based on program analysis. To achieve this, we performed a comparison with PyEvolve, the leading tool for automating unseen variations. Our assessment focused on quantifying the additional opportunities detected and transformed by *PyCraft* in comparison to the baseline.

RQ6. How useful are the generated program transformations? We want to determine whether developers find code improvements generated by *PyCraft* to be useful. To achieve this, we submit pull requests to open-source projects with the patches generated by *PyCraft*.

4.1 RQ1: Effectiveness of Variant Generation

Numerous LLMs have been developed, each with its own set of capabilities and applications. Among them, several of the largest known LLMs include: (i) PALM [1], (ii) GPT-3.5 [9], and (iii) GPT-4 [42]. are prominent examples of the largest LLMs developed by Google and OpenAI, boasting trillions of parameters. The efficacy of *PyCraft* hinges on the choice of the LLM employed. Hence, to assess the effectiveness of generating variants using a selected LLM corpus, we have chosen the aforementioned LLM models. These models are distinguished by their extensive training on a vast number of tokens, comprising billions, and the significant number of parameters they encompass. Furthermore, their internal APIs offer convenient accessibility, an essential factor for seamless integration with *PyCraft*. We conduct both quantitative and qualitative analyses to evaluate the variants generated by them.

4.1.1 Dataset. Dilhara et al. [16] studied a diverse set of top 2,500 *CPATs* in Python ML systems, leading to the discovery of four group of frequently occurring *CPATs* kinds: (i) *dissolve* `for` *loops to domain-specific abstractions* (e.g., Listing 1), (ii) *update API usage* (e.g., `np.dot(np.dot(A,B),C) -> np.linalg.multi_dot(A,B,C)`), (iii) *transform to context managers* (`open("file.txt") -> with open ("file.txt")`), and (iv) *use advanced language features* (e.g., Python list comprehension). The authors' survey involving 650 developers further confirmed the developers' strong interest in automating these identified *CPATs* across all four categories. Thus, we select a representative sample of 20 *CPATs* covering all four kinds to comprehensively answer this research question.

4.1.2 Experimental Setup. We initiated each LLM with the prompt for each of the 20 *CPATs*. This generated many raw variants for each CPAT. Then, we executed *PyCraft*'s validation components to thoroughly examine the four-step validation process described in Section 3.1.3, which includes: (i) Syntax validation, (ii) Type validation, (iii) Import validation, and (iv) Semantic validation. For each *CPAT*, we provide a breakdown of variants, showing the count of instances failing each validation. Variants that successfully pass all validations are categorized as correct and are

suitable candidates for further processing. This analysis helps understand LLMs' variant generation capabilities and underscores the importance of static analysis and validation before selecting variants for further processing.

4.1.3 Results. Our initial observations indicated that PALM, GPT-3 and GPT-4 consistently produced a wide array of variants for all *CPATs*. Figure 2 shows the quantity of variants generated by each LLM for every CPAT. Each bar within the chart represents distinct variants, categorized as those with syntax errors, type errors, import errors, semantic mismatches, and, finally, correct variants suitable for subsequent processing. We noticed that all LLMs consistently generated numerous raw variations for all the evaluated *CPATs*. PALM excelled compared to other LLMs in producing a greater quantity of raw variants, with a notable achievement of 584 variants—surpassing the highest count generated by any LLM for a single CPAT. On average, it generated 518 variants per CPAT, outperforming GPT-4, which produced 398 variants, and GPT-3.5, which yielded 461 variants. This highlights the remarkable creativity and capabilities of LLMs in generating many variations.

However, it is worth noting that the relative performance of LLMs varies significantly in terms of generating correct variations. Notably, PALM consistently exhibited a propensity for generating a higher total number of variants across all *CPATs* compared to GPT-3.5 and GPT-4. However, on average, only 5% of these variants remained free from errors and proved to be correct. Conversely, GPT-4, while producing fewer raw variants than PALM and GPT-3, consistently generating a greater number of distinct correct variants than all LLMs. Quantitatively, on average, GPT-4 generated 102% and 555% more correct variants than GPT-3.5 and PALM, respectively. Despite GPT-4's ability to generate the highest number of correct variants, it still produced incorrect variants at a average rate of 65.5%. This underscores the imperative need for filtering techniques, aka "trust but verify".

LLMs excel in generating unseen variants but also produce errors. GPT-4 generates the most correct variants, but its 65.5% error rate emphasizes the need for error identification techniques.

4.2 RQ2: Effectiveness On Test Case Generation

PyCraft utilizes test cases generated by LLM to verify the semantic equivalence between the original input *CPAT* and the generated variations. Therefore, we evaluate both the LLM's proficiency in generating test cases and the quality of these generated test cases. To achieve this, we conduct a two-fold analysis: initially, a quantitative assessment of LLM's test case generation capabilities, followed by an investigation into the quality of these test cases using mutation testing techniques. Mutation testing involves deliberately introducing subtle modifications (mutations) to the *CPAT* and subsequently retesting the generated tests to determine their ability to detect these mutations. This technique is instrumental in identifying tests that may not effectively uncover faults.

4.2.1 Dataset and Experimental Setup. We chose the LLMs detailed in Section 4.1 and prompted them to generate test cases for all 20 *CPATs* considered in Section 4.1. Before selecting the test cases that will help us choose the correct variants, we followed a three-step validation process, elaborated in Section 3.1.3. This process ensured that the test cases meet the following criteria: (i) they are devoid of syntax errors, (ii) they initialize all its variables, and (iii) original *CPAT* pass the test case. Our quantitative analysis is primarily focused towards evaluating test cases based on these steps, identifying those that do not conform to the criteria as erroneous test cases. Furthermore, we conducted mutation testing on the test cases that successfully met all steps. This aids in comprehending the efficacy of the test cases intended for the selection of correct variants. To generate mutants, we used the widely used *mutmut* [32], known for generating Python mutations.

Table 1. Variant generation for each *CPATs*, and the number of transformations.

#	LHS of CPAT	RHS of CPAT	V	V_c	V_u	V_a	T_1	T_2

1	`count = 0`
	`for i in int_list:`
	`  count =i + count`
		`count = np.sum(int_list)`	1185	291	83	50	17	196 (11x)

2	`for k, v in add_dict.items():` `  d[k] = v`	`d.update(add_dict)`	1201	478	119	110	51	201 (4x)

3	`common = []`
	`for i in l1:`
	`  if i in l2 and i not in common:`
	`  common.append(i)`
		`common = list(set(l1).`
		`  intersection(l2))`
			782	287	107	66	10	141 (14x)

4	`for idx, item in enumerate(values):`
	`  if idx != 0:`
	`  string += ", "`
	`  string += item`
		`string = +", ".join(values)`	285	101	20	10	2	12 (6x)

5	`d = {}`
	`for i in array:`
	`  if i in d:`
	`  d[i].append(f(i))`
	`  else:`
	`  d[i] = [f(i)]`
		`d = {}`
		`for i in array:`
		`  d.setdefault(i, []).append(f(i))`
			1265	416	150	75	9	125 (14x)

6	`counts = {}`
	`for i in iterable:`
	`  if i not in counts:`
	`  counts[i] = 0`
	`  counts[i] += 1`
		`counts = Counter(iterable)`	927	425	202	85	11	106 (10x)

7	`cum_arr = []`
	`for i in range(len(array)):`
	`  cum_arr.append(sum(array[:i+1]))`
		`cum_arr = np.cumsum(array)`	1223	290	95	80	3	68 (23x)

8	`dot_prod = 0`
	`for i in range(len(arr1)):`
	`  dot_prod += arr1[i] * arr2[i]`
		`dot_prod = np.dot(arr1, arr2)`	177	28	26	24	16	208 (13x)

9	`result = []`
	`for i in range(len(array1)):`
	`  result.append(array1[i] + array2[i])`
		`result = np.add(array1, array2)`	64	11	11	9	5	36 (7x)

10	`t = []`
	`for i in range(len(elem)):`
	`  if cond(elem[i]):`
	`  t.append(elem[i])`
		`t = [elem[i] for i in range(len(elem))`
		`  if cond(elem[i])]`
			955	453	226	71	23	907 (39x)

V: Number of unseen variations generated by LLM, V_c: Number of correct variations, V_u: Number of useful variations, V_a: Number of applicable variations, T_1: Number of application performed by PyEvolve, T_2: Number of applications performed by *PyCraft*.

4.2.2 Results. Our observations revealed that PALM, GPT-3, and GPT-4 consistently produced output with test cases for all the input *CPATs*. As shown in Figure 3, we observed that LLMs are capable of generating many unit tests for a given code idiom. In certain cases, it produced up to 503 test cases with GPT-3, resulting in 271 valid test cases among them. GPT-3 outperformed all the LLMs in terms of generating a higher number of test cases. However, it produced test cases with errors, described in Section 4.2.1, at an average rate of 37%. In contrast, GPT-4 generated a relatively lower number of test cases compared to GPT-3 but still more than PALM, and its error rate was significantly lower, standing at 19%. Therefore, GPT-4 produced the highest number of error-free test cases. We observed that GPT-4, on average, created 210 error-free test cases per *CPAT*, ranging from 93 to 372, demonstrating the LLM's capability to generate extensive test suites. This also underscores the importance of employing techniques to select error-free test cases.

We then executed the tools to generate mutants of the *CPATs* and employed the generated test cases to detect both the mutants and the original code. Our observations revealed that the generated test cases achieved a 100% success rate when detecting the mutants generated by the mutation testing tool, *mutmut* [32]. In Section 4.4, we further analyze the effectiveness of these generated tests by comparing their performance to an oracle of human-generated test cases.

LLMs excel in test case generation but also introduce errors. While GPT-4 outperforms others, its 19% erroneous test cases emphasize the importance of validation techniques.

4.3 RQ3: Best Performing Parameters for Generating Variants

The LLM's output varies with temperature (T), prompt iteration (I_p), and feedback iteration (I_f), as they impact randomness and iterative exploration, leading to variations in the generated responses (see Section 3.1.2 for parameter details). Therefore, we adopt an empirical approach to determine optimal values for generating variants. We first generated an oracle of unseen variants consist with

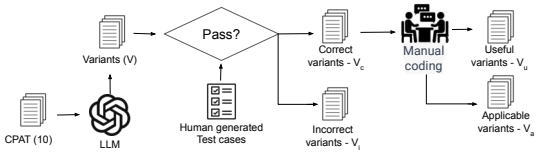

Fig. 4. Schematic diagram of the workflow for generating data to fine-tune LLM parameters

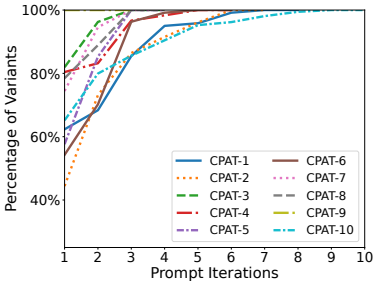

Fig. 5. Generation of variants (Y-axis) along with prompt iteration (X-axis)

9325 unseen variants for 10 *CPATs* given Table 1. Then, we used both automated and manual steps to group the variants into *Correct variants* (Definition 3.2), *Useful variants* (Definition 3.3), and *Applicable variants* (Definition 3.4). Then, we study the generation of these variants in relation to the parameters to determine the best-performing settings for *PyCraft*.

4.3.1 Creating an Oracle. As shown in Figure 4, we invoke LLM to generate unseen variants for the 10 CPATs shown in Table 1. Our objective is to generate a comprehensive set of all possible unseen variants for each CPAT. We generate 10 such sets, and each set is denoted as V^m, where m corresponds to the CPAT number ($m \in \{1, 2, \ldots, 10\}$). To create V^m, we prompted LLM with *CPAT m* for each temperature value in the set $\{0, 0.3, 0.5, 0.7, 0.9, 1.2\}$. For each temperature value, we performed prompt iterations (I_p) up to 15 times, by repeatedly prompting the LLM with the same CPAT. After completing all 15 prompt iterations, we employed feedback iteration, inputting each generated variant back into the LLM. We repeated these steps until no new variants were generated. Through this process, a total of 8064 distinct variants were generated for all CPATs.

As the next step, we statically validated the generated variants, as described in Section 3.1.3. Further, two authors of the paper manually wrote unit test cases for the *CPATs* to evaluate their functionality and consider boundary cases. Then, we executed the test cases on the generated unseen variants to classify them as either correct (V_c^m) or incorrect (V_i^m). This classification process resulted in total 5284 incorrect variants ($\sum_{m=1}^{10} V_i^m$) and 2780 correct variants ($\sum_{m=1}^{10} V_c^m$).

To categorize the correct variants (V_c^m) into "Usable" (Definition 3.3), referred to as (V_u^m), we utilized the Inter-Rater Reliability (IRR) methodology [36]. Following best practices and qualitative research guidelines, the authors employed a negotiated agreement technique [10, 64] to achieve consensus. Two authors of the paper conducted manual analyses on each change pattern to identify the high-level programming tasks associated with them. They reached a consensus to classify a variant as “usable” if it represented something a developer would realistically write, and they labeled variants as “not-useful” if developers would not typically include them in their code. Then, we conducted the static analysis given in Section 3.1.5 to further identify the “applicable” variations (V_a^m) from the “useful” variants. With this process, we identified a total of 1039 useful variants ($\sum_{m=1}^{10} V_u^m$) and 580 applicable variants ($\sum_{m=1}^{10} V_a^m$). This data set serves as a resource for fine-tuning variables during the generation of test cases and variants as explained in the following sections.

4.3.2 Prompt Iteration (I_p). We study the number of prompt iterations required for each temperature value until it no longer produces a significant number of new variants, which guides our decision-making regarding the suitable number of prompt iterations for subsequent steps.

Dataset and Experimental setup: We input the CPATs from Table 1 into the LLM alongside a prompt for variant generation, iterating with the same prompts labeled as i_p^x from the 1st to the 100th iteration, where x represents the iteration number. Then, we study the cumulative count of distinct variants produced during each iteration.

Results: Figure 5 shows the process of variant generation for the initial 10 iterations at temperature 0. In the first iteration, the LLM generated an average of 70% (with a minimum of 45% and a maximum of 82%) of total variants. By the second iteration, this increased to 83% (with a minimum of 68% and a maximum of 96%) of variants. During the third iteration, the LLM generated 95% (with a minimum of 85% and a maximum of 100%) of variants. Beyond that point, we observed that the LLM hardly generates any new variants for a given *CPAT*.

To statistically analyze the data, we started with forming distributions as $D_i^t = \{d_x \mid d_x$ *represents the proportion of total variants generated at the* t^{th}*temprature, by the completion of the* i^{th}*prompt iteration* for the x^{th} *CPAT*, $\forall x \in \{1, 2, \ldots, n\}\}$. We then used the *Wilcoxon Signed-Rank Test* to analyze the paired samples (D_i^t, D_{i+1}^t), $\forall t$. This analysis focused on iterations i within the set $\{1, 2, 3, 4, 5\} \subset \mathbb{N}$, which were marked by significant variant generation. The null hypothesis, positing no significant difference between the variant observations in pairs (D_i^t, D_{i+1}^t) for all i and t, was rejected for $i < 3$ and $t \neq 0$. To quantify the differences between these distributions, we utilized the *Hodges-Lehman estimator*, revealing differences as follows: for $i \in \{1, 2, 3, 4\}$, $D_{i+1}^t - D_i^t = \{25\%, 16\%, 0\%, 0\%\}$ respectively where $t = 0.5$, indicating a potential for increased variant generation with each iteration up to the third iteration. However, this incremental benefit was noted to diminish after the third iteration. This observation is consistent for all t except $t = 0$. At $t = 0$, the randomness is reduced to zero, resulting in the LLM conservatively generating a limited number of variants, all of which are produced at $i = 0$ without extending beyond that point. Consequently, to optimize the trade-off between the merits of additional variant generation and the implications of extended processing duration, we opted to cap the iteration count at three.

Continuing prompt iterations indefinitely is an option to generate numerous variants. However, beyond the third iteration, LLM produces new variants sparingly.

4.3.3 Feedback Iteration (I_f). In order to generate as many variants as possible, we perform feedback iteration by inputting an applicable variant generated in the previous step to the LLM. While this approach can potentially generate many variants, it can also lead to producing numerous not-useful variants (refer Section 3.1.4) when the LLM is prompted with another not-useful variant. This situation can impact the performance of *PyCraft* in two ways: (i) multiple iterations generating not-useful variants, and (ii) inferring rules for not-useful variants that will not be applied to the code. We study the generation of useful variants along with the number of feedback iterations.

Dataset and Experimental setup: We use the oracle mentioned in Section 4.3.1 to study this in two ways. (i) We study, for each temperature, the cumulative ratio of non-useful variants to total non-useful variants in the oracle, generated at each feedback iteration. This analysis helps us understand the trends in generating non-useful variants along with feedback iteration. (ii) In addition to minimizing non-useful variants, it is also important to increase the number of useful cases. Hence, we study the generation of useful cases along with iterations and temperature.

Results: The plot in Figure 6 shows the distribution of the ratio of the cumulative number of non-useful variants generated by each iteration relative to the total non-useful variants in the oracle. This analysis is conducted for each *CPAT* listed in Table 1, represented by lines in each plot, and across various temperature values. While Figure 6 presents the plot for only temperature 0, the others showcase similar trends as depicted in the provided plots and are available on our companion website [49]. We noticed a consistent trend across all temperatures and each *CPAT*, where the generation of non-useful variants begins after approximately 3 to 6 feedback iterations, followed by a rapid increase.

To statistically analyze the data, we define distributions as: $F_i^t = \{f_x \mid f_x$ *represents the proportion of total non-useful variants generated at the* t^{th} *temperature by the completion of the* i^{th} *feedback*

iteration for the x^{th}*CPAT*, $\forall x \in \{1, 2, \ldots, n\}\}$. We then applied the Wilcoxon Signed-Rank Test to analyze the paired samples (F_i^t, F_{i+1}^t), $\forall t$. This analysis focused on iterations i within the set $\{2, 3, 4, 5\} \subset \mathbb{N}$, identified as the iterations where significant generation of non-useful variants begins. The test consistently rejected the null hypothesis, suggesting a significant difference between the variant observations in pairs (F_i^t, F_{i+1}^t) for all i and t. To quantify these differences, we employed the Hodges-Lehman estimator, which revealed that for $i \in \{2, 3, 4, 5\}$ and $t = 0.5$, the differences are: $F_{i+1}^t - F_i^t = \{1\%, 2.5\%, 3.6\%, 5.1\%\}$ respectively, indicating an increasing trend in non-useful variant generation starting significantly after the third iteration. Furthermore, to understand whether the production of non-useful variants behaves the same across all temperature values, we applied again the Wilcoxon Signed-Rank Test to analyze the paired samples for all $(F_i^{t_1}, F_i^{t_2})$, where $t_1 \neq t_2$. At each feedback iteration i, no significant differences were observed in the distributions, indicating that temperature values behave consistently in producing non-useful variants.

While we observe that reducing non-useful variants is more achievable with fewer feedback iterations, a trade-off arises because we also aim to increase the generation of useful variants. To address this, we analyzed the cumulative ratio of useful variants generated up to each iteration compared to the total variants generated, across different temperature values. Consistently across all *CPATs*, we observed that temperatures within the middle range, i.e, 0.5 and 0.7, consistently yield useful variants with fewer feedback iterations compared to other temperatures. For example, with four feedback iterations, temperatures 0.5 and 0.7 yield useful variants spanning 61% to 83% for each *CPAT*; this percentage rises to 69% to 88% with five iterations, and the trend persists with further iterations. To statistically analyze the data, we defined distributions as: $U_t^i = \{u_x \mid u_x$ represents the proportion of total useful variants generated at the t^{th} temperature, from the temperature list temp $= \{0, 0.3, 0.5, 0.7, 0.9, 1.2\}$ relative to the total useful variants in the oracle up to the i^{th} feedback iteration for the x^{th} *CPAT*, for all $x \in \{1, 2, \ldots, n\}\}$. Then, we applied the Wilcoxon Signed-Rank Test to analyze the paired samples (U_{t1}^i, U_{t2}^i), where $t1 \neq t2$ for all i, to check for statistically significant differences. This was followed by the Hodges-Lehman estimator to compute the differences in useful variant generation between each temperature value, after which they were ranked according to the estimator. The tests found that temperatures 0.5 and 0.7 produced a higher number of useful variants compared to all other temperature values.

Our goal is to generate more useful variants while minimizing non-useful ones. We determined that performing five feedback iterations and focusing on variations generated at middle temperatures of 0.5 and 0.7 is an optimal approach. Using these settings, we further apply *PyCraft* in Section 4.7 and Section 4.8 to compare its effectiveness against the baseline. Subsequently, we present the generated variants to actual developers for usefulness evaluation.

Medium temperatures (0.5 - 0.7) yield fewer non-useful variants and more useful variants, all while requiring fewer feedback iterations.

4.4 RQ4: Best Performing Parameters for Generating Test Cases

PyCraft generates test cases to identify semantically equivalent variants for the original *CPAT*. We employ LLM with varying temperatures (T) and repeated prompt iterations (I_p) to generate multiple test cases for a single *CPAT*, fostering diversity through the influence of randomness and iterative exploration. We use the oracle generated in Section 4.3.1 to study the effectiveness of test cases produced by *PyCraft*. We then determine the optimal combination of values for T and $I_p$to generate effective test cases.

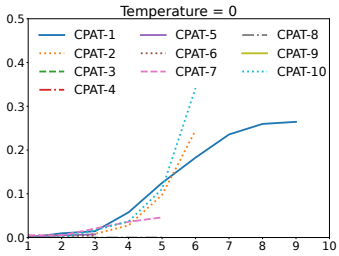

Fig. 6. Generation of not-useful variants along with feedback iterations

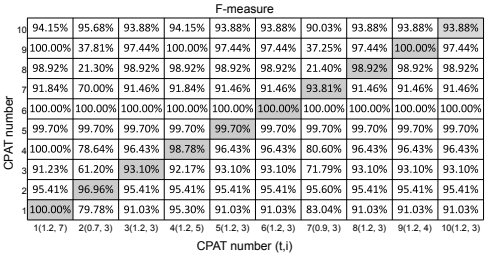

Fig. 7. Cross-validation results for evaluating optimal i and t values to generate test cases

4.5 Dataset and Experimental Setup:

To study the optimal parameters for test case generation, we study the quality of the generated test cases for a range of Temperature (T) values, from the set $\{0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2\}$, and prompt iterations ($I_p = \{1, 2, \ldots, 15\}$). We generate a set of test cases using LLM for each *CPAT*, considering each (t, i) combination where $t \in T$ and $i \in I_p$. These generated test cases are then subjected to the three-step validation process utilized by *PyCraft*. This process enables us to identify and select the valid test cases, denoted as $T^m_{(t,i)}$, across diverse T and I_p values. We then study the effectiveness of testcases, by executing each set of test cases, $T^m_{(t,i)}$, on the oracle V^m (described in Section 4.3.1) and classified them into correct variants $\overline{V}^m_c$ and incorrect variants $\overline{V}^m_i$, and then compare them to the classified variants V^m_i and V^m_c in the oracle. To evaluate the effectiveness of generated test cases in classifying variants as correct or incorrect, we calculated precision and recall. Precision is the percentage of correct variants ($\overline{V}^m_c \cap V^m_c$) among the variants classified as correct ($\overline{V}^m_c$), while recall is the percentage of correctly identified variants among all actual correct variants in the oracle (V^m_c). Then, we compute F-measure using precision and recall values which is the harmonic mean of precision and recall, calculated as $\frac{2 \cdot (\text{precision} \cdot \text{recall})}{\text{precision} + \text{recall}}$.

To identify the most effective combinations of i and t, we adopt the following approach: For each fixed value of i, we calculate the F-measure for all t values. Then, for each i, we identify the t value that yields the maximum F-measure, denoted as $\max_{t \in T} F(t, i)$, along with the corresponding t value, denoted as $t_{\max}$. To further refine this process, we introduce a convergence criterion. If the difference between the maximum F-measure of the current iteration, $F(t_{\max}, i)$, and the maximum F-measure from the previous iteration, $F(t_{\max}, i-1)$, exceeds a threshold δ, we increment i by 1 and repeat the procedure. This iterative refinement goes until the F-measure no longer improves by more than δ. We have chosen δ to be 5% to ensure swift convergence based on the observed data.

After selecting a specific (t, i) combination for V^p that effectively generates test cases to classify the variants, we perform 10-fold cross-validation to assess whether this chosen combination learned from one *CPAT* can be generalized to other CPATs. Let V^m be a set of variations generated for a CPAT, where $m \in 1, 2, 3, \ldots, 10$. After obtaining the optimal combination (t, i) for V^m, we then apply this learned combination to all the other $V^{m'}$s, where $m' \in 1, 2, 3, \ldots, 10$ and $m' \neq m$. This allows us to assess how well this combination generalizes to different patterns.

4.6 Results:

Figure 7 shows the cross-validation results. The x-axis represents the (t, i) configurations that achieved F-measure values for each CPAT index (refer to Table 1 for index to CPAT mapping), while the y-axis represents CPAT index. The outcomes of k-fold cross-validation involve applying the learned (t, i) values to generate test cases with *PyCraft* for other patterns and then calculating their precision and recall values, which are displayed in the respective cells.

The analysis consistently shows that $t = 1.2$ yields higher F-measure, indicating that LLM with a higher degree of randomness is better suited for test case generation. Additionally, the number of iterations varies between 3 and 10 for each CPAT, with higher iterations generally resulting in similar or better precision and recall. However, to strike a balance between effectiveness and efficiency, we choose the value for $i = 5$, which provides superior precision and recall for *CPATs*. The combination of (t, i) strikes a balance between precision, recall, and efficiency, making it the ideal setting for test case generation by *PyCraft* with F-measure 96.6%.

To statistically analyze this, we use the computed F-measure values, $F^i_{\text{temp[t]}}$, for each pattern when detecting correct variants under all the temperature and iteration settings. We formed the distribution, $F^i_{\text{temp}}[t] = \{f_m \mid f_m$ *represents the F1 measure of detecting correct variants for pattern* m^{th} *CPAT where* $m \in \{1, 2, \ldots, 10\}$ at the i^{th} *prompt iteration* $(i \in \{1, 2, \ldots, 10\})$ *and at the* t^{th} *temperature setting from the temperature list* temp $= \{0, 0.3, 0.5, 0.7, 0.9, 1.2\}\}$. We formed 60 distributions for each i, t setting, then selected two distributions at a time and performed the Wilcoxon Signed-Rank Test to analyze the paired samples of F1 scores for each pattern in the two distributions. The test rejected the null hypothesis, indicating a significant difference between the F1 scores across distributions. Subsequently, we applied the Hodges-Lehman estimator to each combination of distributions to quantify the difference in F1 score for each i and t setting. Distributions were then ranked based on the quantifier, revealing that the F1 scores computed for test cases generated at a 1.2 temperature consistently ranked high, with higher iteration values always at the top. However, the difference in estimator values converges after $i = 5$.

Higher temperature (1.2) and more prompt iterations contribute to healthier test cases, resulting in improved F-measure 96.6% classifying correct and incorrect variants.

4.7 RQ5: Effectiveness of *PyCraft*

We conducted a comparative analysis to assess the effectiveness and novelty of *PyCraft* in automating a wider range of code variations, particularly those that were previously challenging to achieve. For this evaluation, we compared *PyCraft* with *PyEvolve*, which is recognized as a leading tool for handling previously unseen variations, with a precision of 97% and a recall of 94% when automating transformations. We applied *PyCraft* to execute CPATs on new target sites and measured its capability to automate unseen variations belonging to both *VT1* and *VT2*. Simultaneously, we used *PyEvolve* with the same input changes to determine its performance in automating unseen variations. This comparison allowed us to assess the impact and uniqueness of our novel contribution in automating a broader spectrum of code variations.

4.7.1 Dataset and Experimental Setup. To employ the *CPATs*, we utilize a selection of 200 top-tier projects that have been previously confirmed by researchers [16] to showcase diversity in aspects such as developers, line counts, Python files, and star ratings. These projects include renowned Python libraries such as *NLTK*, *Keras*, and *microsoft/DeepSpeed*. We configured *PyCraft*'s parameters as discussed in Section 4.3 and 4.4 and applied the *CPATs* listed in Table 1 to these projects. Our analysis focuses on quantifying the target codes that *PyCraft* automated but *PyEvolve* could not, allowing us to comprehensively evaluate *PyCraft*'s unique and advanced capabilities in automating a wider spectrum of code variations compared to *PyEvolve*.

4.7.2 Results. Table 1 shows the applicable variations generated by *PyCraft* for each *CPAT*. It shows that *PyCraft* generates applicable variants, with an average ratio of *58* ranging from a minimum of 9 to a maximum of 110 per CPAT. This empowers *PyCraft* to conduct an average of *58* more searches for applying the *CPAT* compared to *PyEvolve*. Building on this advantage,

Table 1 shows the number of transformations carried out by *PyCraft* for each CPAT, along with those performed by *PyEvolve*. Particularly, *PyCraft* outperforms *PyEvolve* by a minimum of 4x times across all CPATs, achieving a maximum ratio of 39x times transformations compared to the baseline, resulting in an average increase of 14x in transformation opportunities. This demonstrates that *PyCraft* consistently outperforms *PyEvolve* by a substantial margin across all evaluated CPATs. This superior performance is attributed to *PyCraft*'s ability to generate and search for code variants more effectively, resulting in significantly higher opportunities for code improvement compared to the baseline.

PyCraft is capable of generating applicable variations with an average ratio of *58* per CPAT, resulting in an average of 14x more transformations compared to the baseline.

4.8 RQ6: Usefulness of *PyCraft*

We observed that *PyCraft* demonstrates a remarkable ability to automate a broader range of code variations than the baseline *PyEvolve*. However, it is important to examine the usefulness of the code transformations that *PyCraft* automates but *PyEvolve* does not, especially for real-world developers. To accomplish this, we submitted the patches generated solely by *PyCraft* to open-source projects and analyze the response of the developers.

4.8.1 Dataset and Experimental Setup. We selected 40 high-quality projects, including prominent ones like *Keras*, *microsoft/DeepSpeed*, and *NLTK* to apply the *CPATs* listed in Table 1. The inclusion of these professionally-maintained projects underscores *PyCraft*'s ability to identify target codes, which only *PyCraft* can automate, potentially even those that expert programmers might overlook in their coding practices. We executed both *PyCraft* and *PyEvolve*, identifying transformations exclusively performed by *PyCraft* and then submitting them as pull requests to projects.

4.8.2 Results. *PyCraft* effectively transformed 86 instances of *CPATs*, leading to significant improvements in the performance and quality of the affected Python code. These patches updated 124 source code files and impacted 590 SLOC. Following the changes performed in each project, we executed all the available test cases to ensure that the changes did not introduce regressions. Then, we notified the maintainers of the open-source projects through pull requests to incorporate our proposed changes. Our pull requests received acknowledgment and approval even from maintainers of prestigious and extensively optimized codebases, such as those from Microsoft and IBM. In total, we submitted 44 pull requests, each containing 86 instances of *CPATs*. At the time of this writing, 72 (with an acceptance ratio of 83%) instances were accepted, while 4 pull requests were declined. The remaining pull requests are still under review. These positive responses demonstrate the practical usefulness of the program transformations made solely by *PyCraft*.

We discovered three major reasons for pull request rejections: (i) NumPy, a popular library for numerical computing in Python, primarily runs on CPU and does not have native GPU support. Because of that, a pull request submitted to *dmlc/dgl*, which aimed to transform code into NumPy APIs that are optimized for GPUs, was rejected, (ii) A patch submitted to *HKUNLP/UnifiedSKG* was declined because they were hesitant to modify their methods in order to avoid disrupting their forks, and (iii) The project *netsharecmu/NetShare* opted not to accept changes related to a deprecated function that is scheduled for removal in the next release.

PyCraft demonstrated its capability by successfully optimizing code within selected projects that were already highly optimized and well-maintained. This achievement not only showcases *PyCraft*'s considerable value but also emphasizes its ability to discover and leverage substantial opportunities for improvement in less frequently maintained projects. The widespread presence of these projects implies that *PyCraft* has a significant potential to make a meaningful impact on

the broader development community. It offers far-reaching advantages by improving code quality, performance, and maintainability in a diverse range of Python applications.

Developers accepted 83% of the 86 instances, highlighting the usefulness of *PyCraft*'s changes.

5 THREATS TO VALIDITY

(1) **Internal Validity**: *Does our tool produce valid results?* We ensure internal validity by relying on established tools like *R-CPATMiner*, effective for mining *CPATs*, and *PyEvolve*, which achieves 97% precision and 94% recall in code changes.
(2) **External Validity**: *Do our results generalize? PyCraft*'s effectiveness hinges on the chosen LLM model, which we address by evaluating variation generation among top LLMs and selecting the most suitable one. As LLMs upgrade, improved results can be anticipated. The design of *PyCraft* enables seamless adaptation to LLMs. Moreover, the effectiveness of variation generation can vary depending on the selected *CPATs*, potentially limiting the generalizability of the analysis results to other *CPATs*. While this limitation could be mitigated by including more *CPATs*, the manual analysis conducted in evaluations hinders us from adding additional variations. To address this challenge, we selected *CPATs* that cover all four *CPAT* kinds discovered by Dilhara et al. [16].

The techniques in *PyCraft*, designed for Python, are broadly applicable and conceptually robust across different programming languages. Their success in generating code variants largely hinges on the language's syntactic diversity and its library ecosystem's breadth, which provides similar functions via various APIs. Hence, while *PyCraft*'s methods are universally applicable, their performance depends on the language's syntax and library API richness.
(3) **Verifiability**: The data, source code, and executable of *PyCraft* are publicly available [49].

6 RELATED WORK

We group the related work in two areas: (i) Inferring and applying changes using examples changes, and (ii) LLMs for analysing and transforming source codes.

LLMs for analysing and transforming source code: *MELT* [51] is the closest related work that employs LLM to generate transformation examples from library pull requests, facilitating the adaptation of clients to new APIs. While their approach also utilizes LLM, like ours, their primary objective is to address the rule generalization issue [25], aiming for rules that are less specific. In contrast, our focus lies in generating *VT1* and *VT2* unseen variants. Although both approaches tackle two main, distinct challenges in TBE, we hypothesize that combining these two works could further enhance the number of transformations. CodeBERT [23], CodeT5 [62], DeepCoder [6], CodeX [43], EM-Assist [46] and Synchromesh [45] are examples of LLM-based models that have shown promise in tasks such as code completion, code summarization, code refactoring and code generation. Additionally, LLMs have been used for code repair and bug detection, with tools like DeepBugs [48], and MMAPR [66] automatically identifying and fixing common programming errors. Although these techniques have shown efficiency in various facets of source code transformations, our research concentrates on leveraging LLMs to broaden the applicability of TBE systems and transform new target code sites that are variants of the examples found in training. This significantly improves the code transformation potential of TBE systems.

Inferring and applying changes using examples changes: Researchers have made significant improvements in the field of TBE [4, 14, 20, 33, 35, 37, 38, 52, 65]. These techniques employ various approaches to increase the number of reliable code transformations per rule. Lase [38], ReFazer [52], Spdiff [4], TCInfer [33], AppEvolve [20], and APIFix [25] utilize multiple examples to generate transformation rules, identifying commonalities and differences among input examples to abstract adaptations. The effectiveness of these rules increases with the number of input examples

used to infer them. In this regard, APIFix [25] leverages existing codes to synthetically generate example code changes. These changes are then used as inputs to synthesis tools, enhancing the generalization of the rules. On the other hand, PyEvolve [14] and SpInfer [55] are the tools most closely related to *PyCraft*. SpInfer [55] has the ability to handle control flow variants by utilizing the "..." operator to represent random code within the input code. On the other hand, PyEvolve [14] can accommodate data and control flow unseen variants. However, it is unable to handle the more complex *VT1* or *VT2* variants, which can be successfully transformed by *PyCraft*.

Automating Python code changes: Researchers have studied Python idioms and their usage in Python systems. Phan-udom et al. [44] analyzed 58 non-idiomatic and 55 idiomatic changes, while Alexandru et al. [2] provided a list of Python idioms from a developer survey. Sakulniwat et al. [54] studied the evolution of Python's `with` statements over time, while Wang et al. [61] examined Python code smells. Recently, Zhang et al. [67] employed AST rewriting to automatically refactor nine Python idioms. Despite these contributions, the literature tends to offer guidelines and insights rather than complete, automated solutions for such transformations, often limited by the constraints of predefined, static rules. By using *PyCraft*, we implement a technique to infer transformation rules and automate the process of identifying and implementing syntactically different yet semantically equivalent code transformations. *PyCraft* is poised to significantly benefit both Python and ML developers, who form a substantial portion of the Python community [8, 15, 59].

7 CONCLUSIONS AND FUTURE WORK

To accelerate the software development process, it is essential to automate code changes (*CPATs*). Given the wide variety of *CPATs*, we can not encode the analysis and transformation for each *CPAT*, and instead we turn to using TBE systems. So far, progress on using TBE systems to automatically apply *CPATs* has been stifled by the requirement that new target codes exhibit syntax, data-, and control-flow similar to the original *CPAT*. Our novel approach and tool, *PyCraft*, pioneers the synergistic integration of LLMs with TBE systems, and leverages the LLMs' strengths to provide breakthroughs for previous limitations.

Prior to our recent advancements, TBE systems would mine *CPATs* from aging code bases, only to fall prey to software evolution that would render these *CPATs* obsolete and less applicable over time. During our extensive evaluation, we observed that *PyCraft* successfully transformed even target codes that employ the latest language constructs, coding idioms, and library versions. To do this, it generates code variants that employ constructs and features not seen during *CPAT* mining. Achieving this level of compatibility was unfeasible with earlier tools. By staying *perpetually young* and up to date, *PyCraft* transcends software evolution and can revolutionize code change automation.

We anticipate significant usage of LLMs for software engineering (SE) tasks in the coming years. Using *PyCraft*, we demonstrated this potential in code change automation, achieving an increase in code transformations of 14x over the baseline. Moreover, our research confirmed LLMs' creativity *and* their propensity to hallucinate. This highlights the need for targeted research and development of validation techniques to make our SE tools immune to LLM hallucinations. We hope that the techniques we developed in this research inspire others to build upon and move the field forward.

ACKNOWLEDGEMENTS

We thank the ML Methods in Software Engineering Lab at JetBrains Research, and the FSE-2024 reviewers for their insightful and constructive feedback for improving the paper. This research was partially funded through the NSF grants CNS-1941898, CNS-2213763, and the Industry-University Cooperative Research Center on Pervasive Personalized Intelligence.

REFERENCES

[1] Google AI. 2023. Google Bard: An Early Experiment with Generative AI. https://ai.google/static/documents/google-about-bard.pdf

[2] Carol V. Alexandru, José J. Merchante, Sebastiano Panichella, Sebastian Proksch, Harald C. Gall, and Gregorio Robles. 2018. On the usage of pythonic idioms. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2018)*. https://doi.org/10.1145/3276954.3276960

[3] Miltiadis Allamanis and Charles Sutton. 2014. Mining Idioms from Source Code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/2635868.2635901

[4] J. Andersen and J. L. Lawall. 2008. Generic Patch Inference. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE.2008.44

[5] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* OOPSLA (2019). https://doi.org/10.1145/3360585

[6] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *ArXiv* (2016).

[7] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/2635868.2635898

[8] Houssem Ben Braiek, Foutse Khomh, and Bram Adams. 2018. The open-closed principle of modern machine learning frameworks. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1145/3196398.3196445

[9] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[10] John L Campbell, Charles Quincy, Jordan Osserman, and Ove K Pedersen. 2013. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological Methods & Research* 3 (2013). https://doi.org/10.1177/0049124113500475

[11] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2022. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering* 12 (2022). https://doi.org/10.1109/TSE.2021.3128234

[12] James Coplien. 1992. Advanced C++ Programming Styles and Idioms. Addison—W esley. *Reading, MA* (1992).

[13] Barthélémy Dagenais and Martin P. Robillard. 2011. Recommending Adaptive Changes for Framework Evolution. *ACM Trans. Softw. Eng. Methodol.* 4 (2011). https://doi.org/10.1145/2000799.2000805

[14] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. PYEVOLVE: Automating Frequent Code Changes in Python ML Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE48619.2023.00091

[15] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding Software-2.0: A Study of Machine Learning Library Usage and Evolution. *ACM Trans. Softw. Eng. Methodol.* 4 (2021). https://doi.org/10.1145/3453478

[16] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering Repetitive Code Changes in Python ML Systems. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510225

[17] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2568225.2568295

[18] Aleksandra Eliseeva, Yaroslav Sokolov, Egor Bogomolov, Yaroslav Golubev, Danny Dig, and Timofey Bryksin. 2023. From Commit Message Generation to History-Aware Commit Message Completion. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE56229.2023.00078

[19] Zhiyu Fan and Xiang Gao. [n. d.]. Automated Repair of Programs from Large Language Models. ([n. d.]).

[20] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-Usage Update for Android Apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3293882.3330571

[21] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2020. APIMigrator: An API-Usage Migration Tool for Android Apps. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. https://doi.org/10.1145/3387905.3388608

[22] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples *(PLDI)*. https://doi.org/10.1145/3062341.3062351

[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of*

the Association for Computational Linguistics: EMNLP 2020. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[24] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, Hongyu Zhang, and Michael R. Lyu. ASE 2023. What Makes Good In-context Demonstrations for Code Intelligence Tasks with LLMs?. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM. https://arxiv.org/abs/2304.07575

[25] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021. APIfix: Output-Oriented Program Synthesis for Combating Breaking Changes in Libraries. *Proc. ACM Program. Lang.* OOPSLA (2021). https://doi.org/10.1145/3485538

[26] Stefanus A. Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. 2020. Automatic Android Deprecated-API Usage Update by Learning from Single Updated Example. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC)*. https://doi.org/10.1145/3387904.3389285

[27] Stefanus A. Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller. 2022. AndroEvolve: Automated Android API Update with Data Flow Analysis and Variable Denormalization. *Empirical Software Engineering* 3 (2022). https://doi.org/10.1007/s10664-021-10096-0

[28] Stefanus A. Haryono, Ferdian Thung, David Lo, Julia Lawall, and Lingxiao Jiang. 2021. MLCatchUp: Automated Update of Deprecated Machine-Learning APIs in Python. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.1109/ICSME52107.2021.00061

[29] Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/1062455.1062512

[30] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the Naturalness of Software. *Commun. ACM* 5 (2016). https://doi.org/10.1145/2902362

[31] André Hora, Nicolas Anquetil, Stéphane Ducasse, and Marco Tulio Valente. 2013. Mining system specific rules from change patterns. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. https://doi.org/10.1109/WCRE.2013.6671308

[32] Anders Hovmöller. 2023. mutmut - python mutation tester. https://github.com/boxed/mutmut.

[33] Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and Applying Type Changes. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510115

[34] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. 2019. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2019.00072

[35] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2022. A3: Assisting Android API Migrations Using Code Examples. *IEEE Transactions on Software Engineering* 2 (2022). https://doi.org/10.1109/TSE.2020.2988396

[36] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 1 (1977). https://doi.org/10.2307/2529310

[37] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations from an Example. *SIGPLAN Not.* 6 (2011). https://doi.org/10.1145/1993316.1993537

[38] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2013.6606596

[39] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the Fly Synthesis of Edit Suggestions. *Proc. ACM Program. Lang.* OOPSLA (2019). https://doi.org/10.1145/3360569

[40] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2014. Mining Fine-Grained Code Changes to Detect Unknown Change Patterns. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2568225.2568317

[41] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-Based Mining of in-the-Wild, Fine-Grained, Semantic Code Change Patterns *(ICSE)*. https://doi.org/10.1109/ICSE.2019.00089

[42] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[43] OpenAI. 2023. OpenAI Codex. https://openai.com/blog/openai-codex.

[44] Purit Phan-udom, Naruedon Wattanakul, Tattiya Sakulniwat, Chaiyong Ragkhitwetsagul, Thanwadee Sunetnanta, Morakot Choetkiertikul, and Raula Gaikovina Kula. 2020. Teddy: Automatic Recommendation of Pythonic Idiom Usage For Pull-Based Software Projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.1109/ICSME46990.2020.00098

[45] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2021. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *International Conference on Learning Representations*. https://arxiv.org/abs/2201.11227

[46] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Together We Go Further: LLMs and IDE Static Analysis for Extract Method Refactoring. (2024). arXiv:2401.15298 [cs.SE]
[47] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. 2024. EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs. (2024). arXiv:2405.20551 [cs.SE]
[48] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* OOPSLA (2018). https://doi.org/10.1145/3276517
[49] PyCraft-Authors. 2023. *PyCraft*. https://pycrafttool.github.io Accessed: 2024-02-26.
[50] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 8 (2019).
[51] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2023. MELT: Mining Effective Lightweight Transformations from Pull Requests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE56229.2023.00117
[52] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2017.44
[53] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2884781.2884877
[54] Tattiya Sakulniwat, Raula Gaikovina Kula, Chaiyong Ragkhitwetsagul, Morakot Choetkiertikul, Thanwadee Sunetnanta, Dong Wang, Takashi Ishio, and Kenichi Matsumoto. 2019. Visualizing the Usage of Pythonic Idioms Over Time: A Case Study of the with open Idiom. https://doi.org/10.1109/IWESEP49350.2019.00016
[55] Luis Serrano, Vu Anh Nguyen, Ferdian Thung, Lianghao Jiang, David Lo, Julia Lawall, and Gilles Muller. 2020. SPINFER: Inferring Semantic Patches for the Linux Kernel. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association. https://www.usenix.org/conference/atc20/presentation/serrano
[56] Feng Sidong and Chen Chunyang. 2024. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*.
[57] Rishabh Singh. 2016. BlinkFill: Semi-Supervised Programming by Example for Syntactic String Transformations. *Proc. VLDB Endow.* 10 (2016). https://doi.org/10.14778/2977797.2977807
[58] Reudismam Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2021. Learning Quick Fixes from Code Repositories *(SBES)*. https://doi.org/10.1145/3474624.3474650
[59] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE43902.2021.00033
[60] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3314221.3314589
[61] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2021. PyNose: A Test Smell Detector For Python. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE51524.2021.9678615
[62] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. https://doi.org/10.18653/v1/2021.emnlp-main.685
[63] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.
[64] David Wicks. 2017. The coding manual for qualitative researchers. *Qualitative research in organizations and management: an international journal* (2017). https://doi.org/10.1108/QROM-08-2016-1408
[65] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. https://doi.org/10.1109/ICPC.2019.00052
[66] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876* (2022).
[67] Zejun Zhang, Zhenchang Xing, Xin Xia, Xiwei Xu, and Liming Zhu. 2022. Making Python code idiomatic by automatic refactoring non-idiomatic Python code with pythonic idioms. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3540250.3549143

Received 2023-09-29; accepted 2024-01-23