LLM-Guided Genetic Improvement: Envisioning Semantic Aware Automated Software Evolution

Karine Even-Mendoza
King's College London, UK

Alexander Brownlee
University of Stirling, UK

Alina Geiger
JGU Mainz, Germany

Carol Hanna
University College London, UK

Justyna Petke
University College London, UK

Federica Sarro
University College London, UK

Dominik Sobania
JGU Mainz, Germany

***Abstract*—Genetic Improvement (GI) of software automatically creates alternative software versions that are improved according to certain properties of interests (e.g., running-time). Search-based GI excels at navigating large program spaces, but operates primarily at the syntactic level. In contrast, Large Language Models (LLMs) offer semantic-aware edits, yet lack goal-directed feedback and control (which is instead a strength of GI).**

As such, we propose the investigation of a new research line on AI-powered GI aimed at incorporating semantic aware search. We take a first step at it by augmenting GI with the use of automated clustering of LLM edits.

We provide initial empirical evidence that our proposal, dubbed *PatchCat*, allows us to automatically and effectively categorize LLM-suggested patches. *PatchCat* identified 18 different types of software patches and categorized newly suggested patches with high accuracy. It also enabled detecting *NoOp* edits in advance and, prospectively, to skip test suite execution to save resources in many cases. These results, coupled with the fact that *PatchCat* works with small, local LLMs, are a promising step toward interpretable, efficient, and green GI.

We outline a rich agenda of future work and call for the community to join our vision of building a principled understanding of LLM-driven mutations, guiding the GI search process with semantic signals.

***Index Terms*—Large language models, Genetic improvement**

I. Introduction

Large language models (LLMs) are reshaping automated software improvement by enabling the generation of high-level, semantically rich program edits [1]–[6].

In Genetic Improvement (GI), this opens a powerful new source of mutations that go beyond syntax-level rewrites and fixed mutation rules, thus enabling more flexible exploration [5], [6]. However, integrating LLMs as a mutation operator in GI introduces a new challenge: Mutations are often noisy, redundant, or unrelated to the performance goal, commonly leading to *NoOp edits* (no observable behavioral change) or *invalid patch code* [7], [8]. Evaluating each LLM-generated patch through full compilation and testing is expensive and wasteful, and repeatedly generating redundant or similar patches leads to inefficiency and environmental waste [9]–[11]. Hence, while LLMs expand the search space, the process lacks effective guidance to prioritize noteworthy patches. The underlying dynamics of how and why LLM-generated patches work remain poorly understood. We lack a systematic and efficient way to characterize the types of edits produced, how they differ from traditional mutations, and which are most effective in different contexts. Such an understanding is a crucial step towards building solutions in this new AI-powered genetic improvement research agenda: ones that combine LLM creativity with guidance and understanding.

In this paper, we propose a new viewpoint for LLM-guided GI to bridge the gap: The use of a lightweight machine learning model for fast approximation of patch edits as a means to interpret the semantic nature of patches (e.g., "added exception handling", "renamed variable") and guide mutation decisions during search. This model acts as a fast filter, prioritizing promising edits and discarding low-value candidates before costly evaluation, and moving away from blindly sampling patch edits from an LLM's output.

Building on previous work using Sentence-BERT and short-text clustering [12], [13], we introduce *PatchCat* (see §II), trained on manually tagged LLM-generated GI patches of open source projects in Java [6] to group patch summaries. Using *PatchCat*, we characterize the nature of these edits and aim to identify patterns in successful mutations and map the space of effective LLM-driven transformations.

We provide initial experimental results (see (§III)) showing the ability of *PatchCat* to successfully predict *NoOp edits* within the 18 discovered categories of patch edit types, derived from an initial 23 through manual tagging. In this process, two categories were merged and four were excluded for insufficient data. We examined their frequency and quality, with our prototype achieving $\sim$66% accuracy on a validation set of unseen data, which can be further improved with additional steps of data cleaning, category expansion, and enhanced supervision.

This exploratory analysis sets the stage for future work on using structural insights from patch clustering to inform mutation filtering, guide search more effectively, or even steer LLMs toward generating higher-quality patches from the outset.

In addition to this empirical evidence, we present a comprehensive research agenda (see §IV) toward the ultimate goal of AI-powered genetic improvement.

PatchCat, the intermediate trained models, code and training scripts, and datasets used in this study are available online as an open-access artifact [14].

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00339

II. Semantic Classifier for LLM-Guided GI

We evaluate the quality of LLM-generated patches [3], [6], produced via *Gin* [15], a tool designed to facilitate experimentation with GI techniques by automating the process of transforming, building, and testing Java projects. The dataset of patches was generated using an LLM-based mutator with three LLMs on two large-scale projects [16].

Training Phase. (§II-A & Fig. 1a) To analyse these patches, we perform a multi-step process that characterizes their semantic nature and evaluates their impact. Based on this analysis, we build our Automated Patch Classification model, *PatchCat*, to categorize patches into meaningful clusters, representing different types of edits (e.g., control flow changes, comment modifications, API replacements).

Application. (§II-B & Fig. 1b) Using *PatchCat*, we suggest augmenting the *Gin* search process to improve the efficiency of patch evaluation. We implemented *PatchCat* as a light-fast approximation model of the patch's semantics in the training phase, and discuss its integration in the research agenda (§IV).

A. Patch Classification

We outline the creation of *PatchCat* as the training phase in Fig. 1a. To train *PatchCat*, we curated a dataset that contains the *description of a patch* (meaning the difference between the original code version and the patched code) and the category to which we assign the patch. For brevity, we denote the description of a patch as *patchDiff*.

Generate Patch Data. (Fig. 1a, Ⓐ) We generate *patchDiff* for each patch suggested by *Gin* using `diff original_i.java patched_i.java`.

Manually Describe Patches. (Fig. 1a, creating Ⓑ) We manually annotate each *patchDiff* with a *short natural language description*, capturing the essence of the change, producing a paired dataset of *patchDiff* and *briefSum*.

Manually Categorize Description. (Fig. 1a, creating Ⓒ) We manually assign descriptions to categories (e.g., comment deletion, control flow change, API replacement), creating a labeled dataset that contains common types of patch intents. This produces a paired dataset of *Category ID* and *briefSum*. We manually tagged 309 entries, limited to the `Mistral` generated patches, and `JUnit4` and `JCodec` projects (those with the highest pass rates in Brownlee et al. [6], [17]'s work).

Data Augmentation. (Generate more patch descriptions per category, Fig. 1a, creating Ⓓ) For each category, we wrote a small Python program to generate synthetic summaries. This enriches the dataset with labeled, synthetic training data. We manually reviewed the results to confirm they were sensible. This yields a dataset of 11,381 entries, which is reduced to 5,806 unique entries used to train *PatchCat*.

Building *PatchCat*. (Train model with descriptions and categories, Fig. 1a, creating Ⓔ) We used the dataset from the previous phase to train *PatchCat*. After splitting the data of 5,806 unique entries into training and test sets, we applied a semi-supervised clustering approach [13], previously shown to improve short text clustering on StackOverflow question titles [18]. In this work, we adapt the approach in [13] to short descriptions of code patches. We use Python's K-Means [19] as a baseline. We embed all summaries using Sentence-BERT (MiniLM) [12], [20]. We measured accuracy at 0.792 and Normalised Mutual Information (NMI) at 0.735 of the baseline. After enhancing with [13], the cluster alignment was with accuracy = 0.787, NMI = 0.741. The trained model and vectorizer form *PatchCat*.

B. Automation

We outline the usage of *PatchCat* in *Gin* as the application phase in Fig. 1b. The application phase aids the automatic analysis of patches post-execution of *Gin* (as discussed in RQ2 in §III). This phase enables the collection of statistics on patch quality and diversity of generated patches across large codebases. In §IV, we discuss extending this approach to not only automate post-processing but further integrate it as a feedback-guided component within *Gin*.

New Patch. (Fig. 1b, Ⓕ) The process begins with a patch produced by *Gin*. We create a *patchDiff* with the original and the *Gin* -mutated patched code, e.g., Listing 1.

```
312c312,313
< throw new ParseException(``Timestamp '' + timestampStr + '' could not be parsed using a server time of '' +
        serverTime.getTime().toString(), pp.getErrorIndex());
---
> String errorMessage = ``Timestamp '' + timestampStr + '' could not be parsed using a server time of '' + serverTime
        .getTime().toString();
> throw new ParseException(errorMessage, pp.getErrorIndex());
```

Listing 1: *patchDiff* of a patch generated with `Mistral` for the project `Commons-Net`.

Automatically Describing a New Patch. (Fig. 1b, creating Ⓖ) To facilitate further automation, we implemented a script that, given a *patchDiff* generates a short summary of the changes, i.e., *briefSum*, using the `llama3` LLM and the prompt in Listing 2. This summary abstracts away irrelevant syntactic changes and highlights the core modification.

```
``Summarize the following Java diff in exactly 15 words: <<diff>>"
```

Listing 2: Prompt to generate a *briefSum* of *patchDiff*

Using *PatchCat* to Assign Patch to a Category. (Fig. 1b, receiving as input Ⓗ, invoking Ⓘ and outputting Ⓙ) The *briefSum* is then passed to *PatchCat*, which maps it to a predefined semantic category (e.g., *"just added try and catch"* to Category 9 as exemplified in Listing 3).

```
$ ... ``nothing much there is no difference really" . . => [1]
$ ... ``seems like there are only new comments" . . . => [2]
$ ... ``just added try and catch" . . . . . .=> [9]
```

Listing 3: Examples of querying *PatchCat*.

***PatchCat*-Guidance Enhancing AI-powered GI.** (FUTURE WORK dashed box, Fig. 1b, Ⓙ as use case example) This classification could be used in the future to inform downstream decisions.

Example A: NoOp Patches. *No Operation* (*NoOp*) patches do not change the behavior of the program or are dead code. Hence, patches of Categories 1, 2 and 17 (see Table I) are an example of NoOp. In practice, we would like to keep only the patches that compile and pass tests that are not simple *NoOps*. As the patch interpretation in *Gin* will be part of our future work, we will discuss this further along with more complex scenarios in the research agenda in §IV.

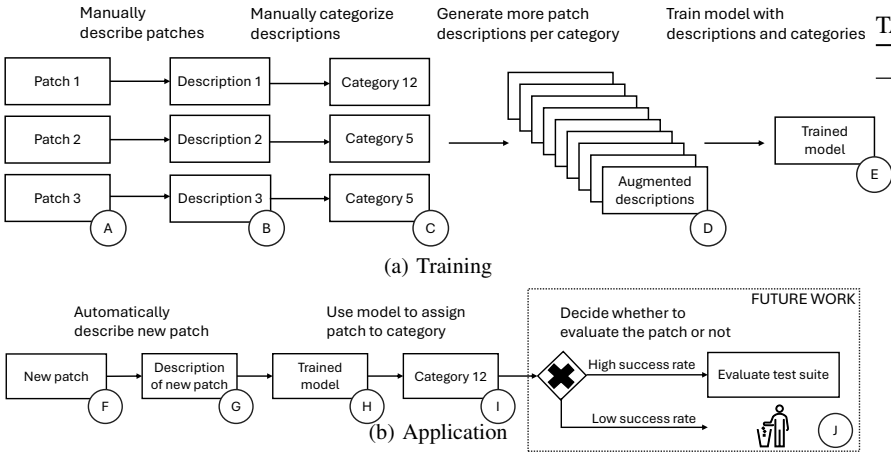

Fig. 1: Training (a) and Application (b) Scenarios. In the training phase, each patch is manually described and categorized. These descriptions are used to generate augmented data to train *PatchCat*. In the application phase, *PatchCat* automatically assigns patches to categories. Only those in promising categories are tested; the rest are discarded.

TABLE I: Category (#) & Description.

#	Description
0	Added (some arbitrary) code from GitHub
1	No change
2	Modified a comment (add/remove/edit)
3	Deleted blocks in a method (all/most/some)
4	Duplicate code
5	Modifications to return statements (add/remove/edit)
6	Changes to method names
7	Changed data types or type usage and generics
8	Includes inlining of implementations (sort, methods...)
9	Added exception-handling constructs (unreachable or reachable)
10	Added extra brackets
11	Added synchronization logic
12	Modified Variable/Class/Object Name
13	Modified Control Flow Structure
14	Modified Object/Primitive Creation or Initialization
15	Split a statement into multiple lines
16	Arithmetic manipulation (boolean var. or expr. manipulations)
17	Added dead code

III. Preliminary Evaluation

Our classified patches are taken from Brownlee et al. [6], [17]'s work, which recorded whether each patch could be *compiled*, whether the resulting patched code still *passed* all unit tests, and whether it was in fact a *NoOp*.

In this work, we evaluate two core aspects: the accuracy of *PatchCat* and the nature of LLM-based patches in *Gin*, each tied to one research question (RQ). Therefore, we ask:

RQ1: *How accurately does the model assign categories to patches based on the patchDiff between original and patched code, and how reliably can these categories inform decision-making in Gin?*

RQ2: *What types of patches does Gin attempt to apply when mutating code via LLM-generated edits, and what is their quality in terms of diversity, static validity (i.e., compilation rate), and dynamic validity (i.e., passing the software under optimisation's test suite)?*

RQ1: Model Validation. We assessed the model's performance using previously **unseen data**. We sampled 218 records from 5 different datasets of 3 real-world software projects [16] [6], none of which was used in §II-A to train or test the model. We automatically constructed a dataset of 218 (*patchDiff*, *briefSum*) pairs. Then, we manually assigned them to the categories in Table I and afterwards independently tagged them with *PatchCat*, resulting in the final format: (*patchDiff*, auto-generated *briefSum*, manually tagged category, auto-tagged category). A cleanup stage was required before invoking *PatchCat* to remove all quotation marks, backticks, the padded phrase *"Here is a 15-word summary:"* or *"Java diff:"*, and to adopt the verbs to fit the GI wordings (e.g., change "update" to "modify"). To evaluate the quality of the model on unseen projects, we compared the manually and automatically assigned categories, out of which 75 did not match, while 143 were consistent. Table II summarises per-category accuracy and overall accuracy. There are 6 categories to which no patch was manually assigned. 15 manual tags were invalid or inconclusive, suggesting a need to improve tagging, descriptions, or clustering granularity.

Example 1: A *briefSum* was *"Java code change: catching ParseException with variable name "e" and added try-catch block"* misdescribed a *patchDiff* with no new try-catch block. The manual tag was left blank, while *PatchCat* assigned Category #9.

Example 2: A *briefSum* was *"Java code modified to handle timezone offset and check for daylight saving time"* matched no existing category and was arbitrarily assigned Category #4.

We analyzed the remaining 60 inconsistencies in Fig. 2, which shows the differences between the manual and automated tags. Each x-axis label denotes an auto–manual pair (e.g., 0–3 means auto-tagged as Category #0, manually as #3); the y-axis shows the count of such cases.. The largest mismatch involved 19 instances manually tagged as Category #1 ("No change") but *PatchCat* assigned them to Category #17 ("Added dead code"), likely reflecting *PatchCat*'s difficulty distinguishing edits with no behavioral effect.

We identified possible causes for the inconsistencies: **Missing categories**, as shown in Example 2; **Different prioritization** by *PatchCat* and humans, e.g., a case was tagged #12 by *PatchCat* and #9 by a human who noted: *"12 could also be considered but 9 is more important..."* for *briefSum*: *"A Java code diff with 4 changes: catches ParseException, adds variable, and updates logic"*; and **Incomplete or unclear summaries**, see example in [14].

RQ1 Answer. On unseen data from five datasets, *PatchCat* achieved 66% accuracy, lower than on test data in §II-A. For generalization, there is a need to cleaning LLM-generated text before model input, adapting new categories for new projects, and adding more manually tagged data to reduce misprioritisation, to enhance *PatchCat*'s reliability for decision-making in *Gin*.

RQ2: Patches Quality. We used *PatchCat* on a dataset of

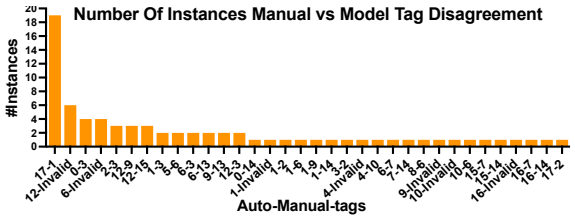

Fig. 2: Mismatch Frequencies Manual Vs. Auto Patch. X-axis shows category pairs `a--b`: `a` is an automatically predicted category by the model & `b` is the category assigned manually.

TABLE II: Per-category Clustering Accuracy & Category Size.

Cat.	0	1	2	3	4	5	6	7	8
Acc.	NA	0.85	0.50	0.00	NA	NA	0.00	0.00	1.00
Size	0	129	6	13	0	0	5	3	1
Cat.	9	10	11	12	13	14	15	16	17
Acc.	0.78	0.50	NA	1.00	0.20	0.55	0.40	NA	NA
Size	18	2	0	5	5	11	5	0	0

Overall accuracy: 0.66 with 218 manually tagged instances.

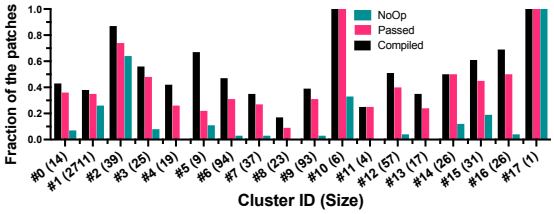

Fig. 3: Patch Categorizing Results. X-axis label per Category ID (its size) & Y-axis is the fraction of patches compiled, passed all unit tests and did not perform any operation (NoOp).

3,232 patches from [3], [6], for 5 projects and generated by 3 LLM models [16]. Further, we report the number of patches that *compile*, *passed* all tests or are *NoOps* in each category.

Fig. 3 summarizes the patch statistics of the automatically assigned patches. *Compiled* and *Passed* report the number of patches per category that compiled, and passed all tests, respectively. *NoOp* indicates the proportion of semantically neutral patches, i.e., those with no observable effect.

The most frequent category is "No change" (Category 1, 2,711 patches), followed by much smaller categories: 94 patches are in "Changes to method names" (Category 6), 93 in "Added exception-handling constructs (unreachable or reachable)" (Category 9), and 57 in "Modified Variable/Class/Object Name" (Category 12). Except for Categories 1–2 and 17, there should be no NoOps. Manual inspection of misclassified cases revealed misleading *briefSum*, e.g., *"Special case added to look for partial maps within a list in code"*, despite the switch statement remaining unchanged. Categories 3–16, likely with observable effects, have compilation rates from 0.17 to 1.00. The lowest test passing rate is 0.09 (Category 8, 0.17 compilation rate). The sharpest relative drop appears in Category 5, where only 0.22 of the patches passed all tests despite a 0.67 compilation rate, i.e., just one-third of the compiled patches were successful. The high rate of NoOp failures is attributed to this class including approximated 37% of suggested patches that included no code, so no diff, and were counted as fails in [16]. In a GI context, one would prefer patches in categories like 7 and 16, where the fraction of passing tests is high but the number of NoOps is low: these are patches that change the code implementation in some way that preserves functionality (at least with respect to the tests). This can form the basis of a filtering scheme as we propose in our research agenda (§IV).

RQ2 Answer. The high rate of NoOp patches suggests that filtering them using *PatchCat*, a lightweight alternative to LLM calls, can improve *Gin*'s performance. Moreover, patches that compile and have an observable effect tend to have a good chance of passing the test suite, making them valuable in the context of GI.

IV. Conclusion & Research Agenda

We introduced *PatchCat* for classifying automatically generated software patches and to gain deeper insights into the nature of LLM-based edits. *PatchCat* classified patches into 18 different categories and achieved 66% accuracy for patches generated for five unseen real-world software projects. Moreover, *PatchCat* enabled us to detect *NoOp* edits in advance and, prospectively, skip the execution of the test suite in many situations to save resources.

We envisage future work to leverage the insights from our work to further improve *PatchCat*, for example, one can train *PatchCat* with patches generated for additional software projects to have a broader and more diverse data base so that the classifier is optimally equipped for practical usage. Further, we will study the usage of LLMs to improve the classifier [21]. In addition to LLM-based edits, the improved classifier can be used to analyze traditional mutation methods and compare the results with the LLM-generated patches in order to learn when a specific edit type is more useful or when a combination is more beneficial. Our expectation is that while LLM-generated patches are often semantically more adequate, traditional mutations introduce more diversity [22], which can lead to solutions that would not have been possible with LLMs alone. A combination could unlock the advantages of both methods for GI and can be efficiently obtained by using appropriate heuristics and ML methods [23]. Furthermore, *PatchCat* can be used to improve the search for patches with *Gin*. To this end, one can implement Fig. 1b - step Ⓙ which classifies the patches generated by *Gin* during the run. Based on the assigned class, it can be decided whether an evaluation with the test suite should be performed. For patch classes that have shown a high success rate so far, it is worth running the test suite. For classes with a low success rate, e.g., the evaluation can be skipped for *modified a comment* or *added dead code*. Given *PatchCat's* ability to detect *NoOp* edits in advance, we expect to save a significant amount of compute for the evaluation in *Gin*. Additionally, one can provide the user with helpful information in addition to the suggested patches. Last but not least, explanations for the respective patches [2] can be automatically generated to describe what led to performance improvements.

REFERENCES

[1] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, " Large Language Models for Software Engineering: Survey and Open Problems ," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 31–53. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE-FoSE59343.2023.00008

[2] D. Sobania, A. Geiger, J. Callan, A. Brownlee, C. Hanna, R. Moussa, M. Z. López, J. Petke, and F. Sarro, "Evaluating explanations for software patches generated by large language models," in *International Symposium on Search Based Software Engineering*. Springer, 2023, pp. 147–152.

[3] A. E. I. Brownlee, J. Callan, K. Even-Mendoza, A. Geiger, C. Hanna, J. Petke, F. Sarro, and D. Sobania, "Enhancing genetic improvement mutations using large language models," in *Search-Based Software Engineering - 15th International Symposium, SSBSE 2023, San Francisco, CA, USA, December 8, 2023, Proceedings*, ser. Lecture Notes in Computer Science, P. Arcaini, T. Yue, and E. M. Fredericks, Eds., vol. 14415. Springer, 2023, pp. 153–159. [Online]. Available: https://doi.org/10.1007/978-3-031-48796-5_13

[4] X. Wu, S.-H. Wu, J. Wu, L. Feng, and K. C. Tan, "Evolutionary computation in the era of large language model: Survey and roadmap," *Trans. Evol. Comp*, vol. 29, no. 2, p. 534–554, Nov. 2024. [Online]. Available: https://doi.org/10.1109/TEVC.2024.3506731

[5] J. Wang, C. Hanna, and J. Petke, "Large language model based code completion is an effective genetic improvement mutation," in *2025 IEEE/ACM International Workshop on Genetic Improvement (GI)*. IEEE, 2025, pp. 11–18.

[6] A. E. I. Brownlee, J. Callan, K. Even-Mendoza, A. Geiger, C. Hanna, J. Petke, F. Sarro, and D. Sobania, "Large language model based mutations in genetic improvement," *Autom. Softw. Eng.*, vol. 32, no. 1, p. 15, 2025. [Online]. Available: https://doi.org/10.1007/s10515-024-00473-6

[7] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4All: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[8] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen, "On the evaluation of large language models in unit test generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1607–1619. [Online]. Available: https://doi.org/10.1145/3691620.3695529

[9] V. Campos, R. Shariffdeen, A. Ulges, and Y. Noller, "Empirical evaluation of generalizable automated program repair with large language models," 2025. [Online]. Available: https://arxiv.org/abs/2506.03283

[10] J. Petke, B. Alexander, E. T. Barr, A. E. Brownlee, M. Wagner, and D. R. White, "Program transformation landscapes for automated program modification using gin," *Empirical Software Engineering*, vol. 28, no. 4, p. 104, 2023.

[11] A. Kiran, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A comprehensive investigation of modern test suite optimization trends, tools and techniques," *IEEE Access*, vol. 7, pp. 89 093–89 117, 2019.

[12] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[13] M. R. H. Rakib, N. Zeh, M. Jankowska, and E. Milios, "Enhancement of short text clustering by iterative classification," in *Natural Language Processing and Information Systems*, E. Métais, F. Meziane, H. Horacek, and P. Cimiano, Eds. Cham: Springer International Publishing, 2020, pp. 105–117.

[14] K. Even Mendoza, A. Brownlee, A. Geiger, C. Hanna, J. Petke, F. Sarro, and D. Sobania, "Artifact of llm-guided genetic improvement: Envisioning semantic aware automated software evolution," Jul. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15834984

[15] A. E. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White, "Gin: genetic improvement research made easy," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 985–993.

[16] A. E. Brownlee, J. Callan, K. Even Mendoza, A. Geiger, C. Hanna, J. Petke, F. Sarro, and D. Sobania, "Dataset of five java projects: `commons-net`, `gson`, `jcodec`, `junit`, and `karate`, and generated by three llm models: `Mistral`, `OpenAI`, and `TinyDolphin`," Aug. 2024, derived from the ASE-Journal-V2 artifact on LLM-based mutations in Genetic Improvement. [Online]. Available: https://doi.org/10.5281/zenodo.13381774

[17] ——, "Artifact of large language model based mutations in genetic improvement (journal version)," Aug. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13381774

[18] J. Xu, B. Xu, P. Wang, S. Zheng, G. Tian, J. Zhao, and B. Xu, "Self-taught convolutional neural networks for short text clustering," *Neural Networks*, vol. 88, pp. 22–31, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608016301976

[19] S. learn developers, "Kmeans — scikit-learn 1.7.0 documentation," https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html, our code invokes scikit-learn KMeans with the option "array-like of shape" to enforce 18 semantic categories. Accessed: 2025-07-08.

[20] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 5776–5788. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[21] J. K. Miller and T. J. Alexander, "Human-interpretable clustering of short text using large language models," *Royal Society Open Science*, vol. 12, no. 1, p. 241692, 2025.

[22] D. Sobania, J. Petke, M. Briesch, and F. Rothlauf, "A comparison of large language models and genetic programming for program synthesis," *IEEE Transactions on Evolutionary Computation*, 2024.

[23] A. Aleti and M. Martinez, "E-apr: Mapping the effectiveness of automated program repair techniques," *Empirical Softw. Engg.*, vol. 26, no. 5, Sep. 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-09989-x