Are AI Agents for Code Robust to Metamorphic Transformations?

Andrei Drăgoi

Supervisors: Annibale Panichella[1], Mitchell Olsthoorn[1], Pouria Derakhshanfar[2]

[1] *EEMCS, Delft University of Technology, The Netherlands*
[2] *JetBrains Research, The Netherlands*

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfillment of the Requirements
For the Master of Data Science and Artificial Intelligence Technology

To be defended on July 14, 2026

Name of the student: Andrei Drăgoi
Final project course: DSAIT5000 Thesis Project
Thesis committee:

Thesis advisor: Dr. Annibale Panichella
Daily supervisor: Dr.ir. Mitchell Olsthoorn
External committee member: Dr.ir. Jie Yang

Are AI Agents for Code Robust to Metamorphic Transformations?

Andrei Drăgoi
Delft University of Technology
Delft, The Netherlands

Abstract

Autonomous coding agents such as Claude Code and Codex are now used broadly to write code across entire repositories. They rely on large language models (LLMs), which are known to be sensitive to small behavior-preserving changes in their input. Renaming a variable or reordering code can reduce the accuracy of a standalone model, even though the program's functionality remains the same. It is not clear whether agents have this same weakness, since they can explore the codebase, run tests, and correct any issues through a feedback loop.

This work studies the robustness of coding agents to metamorphic transformations applied across the repository. We use three behavior-preserving transformations that rename identifiers, restructure control flow, and reorder methods. We transform 25 instances from the SWE-rebench-V2 dataset and run Claude Code and Codex on both the original and transformed versions.

The results show that coding agents appear more robust than standalone LLMs. Still, both agents tend to explore the codebase more in the transformed version. For Claude Code, we see an increase in the success rate, while Codex shows greater run-to-run variability in the input tokens, making it less predictable. We read these trends as evidence that the agents are relying less on memorized patterns and more on freshly-read code. Still, this tendency toward robustness should be considered a practical observation, not definitive proof. The run-to-run variance of the agents is large, often larger than the effect we look for, so we can only show we don't see a difference, but cannot prove there is none. So, to answer the question in our title: *in practice, most likely robust.*

1 Introduction

The continuous progress of Generative Artificial Intelligence (AI) is changing how software developers write code [30]. Large Language Models (LLMs) are used in tasks such as code completion [41], test generation [35, 36], and program repair [38, 43]. Nevertheless, the focus is moving from isolated task completion to the use of autonomous agents that can work on larger problems [14, 20]. Unlike simple conversational interfaces, agentic frameworks such as Claude Code[1] or Codex[2] rely on tools to directly access and edit the entire codebase, providing broader and faster assistance to developers [21]. According to the 2025 Stack Overflow Developer Survey [40], nearly 31% of respondents were already using AI coding agents in 2025.

These agents rely on non-deterministic large language models, so they could be affected by the same problems observed in LLMs, such as memorization [1], or lack of robustness against prompt variations [49] and adversarial attacks [29]. In addition, agents are often evaluated on public datasets that are likely part of the model's training data, and thus assess their ability to remember, rather than to solve [1, 34]. However, because they use a feedback loop, agents may detect and fix their own mistakes, making them more robust to perturbations than standalone LLMs. Therefore, it is important to test whether agents are robust.

Metamorphic testing is a method used to evaluate the robustness of LLMs for code. It applies semantic-preserving transformations, also called metamorphic transformations, and checks for output alterations [3]. Here, robustness is the degree to which the system maintains correctness when its underlying components (e.g., the input data) are perturbed [48]. Studies have shown that metamorphic transformations affect LLMs for code [31, 39]. However, these studies rely on datasets that contain function-level tasks and in which the LLM is prompted with a single, small code snippet, such as HumanEval [10] or CruxEval [19]. To evaluate coding agents effectively, and not just the underlying language models, larger and more challenging problems are required, for example, ones that require multi-file edits. Such datasets with larger tasks suitable for coding agents exist, but they make it more challenging to apply metamorphic transformations. Common transformations, such as identifier renaming [5], can no longer be treated as local changes and must be applied consistently across the entire project to prevent compilation failures.

Despite the utility of metamorphic testing in assessing fairness [13] and robustness [4] of Machine Learning (ML) systems, it is not yet commonly used to assess these characteristics on LLM-based coding agents, with many recent benchmarks focusing mostly on high-level success scores only [47]. This work focuses on the robustness of coding agents by scaling and applying metamorphic transformations to repository-level tasks and evaluating the agents on the resulting instances. In line with most benchmarks, this study looks at success rate to assess robustness, but also considers LLM and tool use. We propose the following research questions to evaluate agent robustness to metamorphic transformations:

RQ1: *What is the impact of metamorphic transformations on AI coding agents, regarding* ***task success rate****?*

RQ2: *What is the impact of metamorphic transformations on AI coding agents, regarding* ***resource and tool usage****?*

RQ3: *How do the metamorphic* ***robustness trends*** *identified in standalone LLMs for code generalize to AI coding agents?*

Our study answers these research questions by analyzing two coding agents: Claude Code and Codex CLI on the *SWE-rebench-V2* dataset [7]. We adapt, expand, and apply metamorphic transformations to a subset of instances, and analyze the agent's predictions and usage metrics relative to those on the original data points.

Agents appear more robust in terms of success rate than standalone LLMs, but we identify trends indicating increased codebase exploration and input token variability. The run-to-run variance of coding agents is large, so while we cannot prove that agents are completely robust, in the practical sense, we observe no general differences when metamorphic transformations are applied. This

[1] https://www.anthropic.com/product/claude-code
[2] https://openai.com/codex/

is different from standalone LLMs, which lose accuracy under the same kinds of transformations. This general trend of robustness does not completely rule out data leakage or memorization. Still, it shows that the agents, using a feedback loop, can adapt to changes in the codebase and fix any potential issues caused by memorization. Our contributions are:

- We present a study on the robustness of two coding agents, Claude Code and Codex, to metamorphic transformations applied on repository-level tasks.
- We show that the loss of accuracy reported for standalone LLMs under these transformations does not appear for either agent. Yet, we identify trends showing increased repository exploration and higher agent variability.
- We find that the run-to-run variance of coding agents is large enough to make this kind of robustness expensive and hard to measure, and we discuss what this means for future evaluations.
- We developed CodeCocoon, a tool for applying metamorphic transformations across Java repositories, and released it publicly together with our evaluation results.

2 Background and Related Work

2.1 LLM-Based Coding Agents

A standalone large language model can produce code in a single turn, based on the given input [11]. Autonomous coding agents are built on top of LLMs and allow them to interact with the codebase and environment through tools [25]. They do this through a feedback loop in which the agent reads the code, takes an action (e.g., editing a file or running a test), observes the result, and uses it to decide what to do next.

SWE-agent [44] shows that this loop-based approach can be used effectively in software engineering. They proposed an agent-computer interface for navigating a repository, editing files, and running tests, which at the time of publication, raised the resolve rate on SWE-bench from 3.8% for non-interactive models to 12.5% [44]. The agents studied in this work, Claude Code and Codex CLI, follow the same loop-based approach [2, 27].

2.2 Metamorphic Testing

Metamorphic testing (MT) was introduced by Chen et al. [12] to generate additional test cases in the absence of a reliable test oracle. Instead of checking a single output against an expected value, MT defines a *metamorphic relation*, a property that relates the inputs and outputs of multiple executions of the program. A transformation is applied to a source input to derive a follow-up input, and a violation of the expected relation between their outputs reveals a fault [37].

This makes metamorphic testing well-suited for systems whose correctness is hard to specify, such as machine learning models. This principle can also be applied to code. Applis et al. [4] apply metamorphic program transformations to assess ML-based program-analysis tools, and in a different work, they use a genetic algorithm to guide these transformations [3]. A recent review on the use of MT for deep code models treats semantic-preserving code transformations as metamorphic relations [5]. So, a robust agent should still produce an equivalent solution even when working with a modified codebase, if its original behavior remains unchanged.

2.3 Robustness

Many studies show that neural code models are sensitive to transformations that preserve program semantics. Yefet et al. [46] introduced adversarial examples for code, showing that renaming variables or inserting dead code can force models such as *code2vec* to make incorrect predictions on up to 94% of perturbed inputs. Rabin et al. [32] look at robustness from a generalizability perspective, and find that semantic-preserving transformations frequently change the predictions of neural program models.

For code generation, ReCode [42] provides a robustness benchmark that perturbs docstrings, identifiers, and syntax, and reports that small perturbations can make code generation models produce very different outputs. The same problem appears in LLMs. Recent studies report that LLMs change their answers under semantics-preserving mutations [28], do not reliably predict execution behavior [39], and are sensitive to the input ordering of code [33]. Robustness studies almost exclusively evaluate standalone LLMs on small, function-level snippets [15, 31]. Whether the same problem also affects coding agents that work on entire repositories is a relatively underexplored question, and this is the gap we address.

2.4 Memorization and Data Contamination

Carlini et al. [9] show that LLMs can reproduce a measurable fraction of their training data verbatim. For code, this directly inflates benchmark scores. Riddell et al. [34] find that models perform much better on contaminated problems than on uncontaminated ones, and traces of memorization persist in LLMs for code [1]. This problem is not limited to function-level benchmarks. Liang et al. [24] report that state-of-the-art models can identify buggy file paths from the issue description alone and reproduce repository functions with high verbatim overlap, indicating benchmark-specific memorization rather than reasoning.

Existing approaches to address this are mostly time-based, with some benchmarks evaluating only on problems released after a model's training cutoff date, to be "contamination-free" [22]. Metamorphic transformations offer an alternative approach that does not depend on dataset recency. By modifying existing tasks while preserving their semantics, they test whether reported capabilities reflect real problem-solving abilities or only memorized solutions.

3 Methodology

This study uses a three-stage pipeline for transformation and evaluation, designed to measure the robustness of coding agents to metamorphic transformations. The stages are: (1) *Applying the metamorphic transformations*, (2) *Creating the transformed dataset* and (3) *Prediction & Evaluation.* A complete pipeline figure is available in Appendix A.1.

3.1 Renaming Transformations

Identifier renaming is the most common transformation category used in metamorphic testing [5]. To evaluate the robustness of autonomous coding agents, our study applies semantic-preserving renaming transformations to Java classes, methods, and variables. Since autonomous coding agents require access to the entire codebase and have tools to edit, compile, and run it, simple text-based substitution cannot handle scope and cross-file references.

Our transformation pipeline, CodeCocoon, is built on the IntelliJ Platform SDK's Program Structure Interface (PSI) to ensure all identifiers, usages, overrides, and imports are renamed consistently throughout the codebase. The transformations use suggestions generated by an LLM to ensure that the new names are plausible given the context, and similar to human-generated ones. The complete source code is publicly available[3].

3.1.1 ***Identifier filtering.*** Not all identifiers in a Java project can be renamed without breaking functionality. Some examples include classes used in databases, canonical methods (*equals*, *toString*, etc.), or getters and setters. Thus, the pipeline filters out some identifiers.

Class renaming. The pipeline excludes from the renaming process classes used in non-Java files, test and generic type classes, and some annotated classes. We exclude classes that are managed by external frameworks, such as the Java Persistence API (e.g., *@Entity*) or the Spring Framework (e.g., *@Controller*). All other types of classes are included (e.g., core domain classes, application-specific logic, or internal data structures).

To ensure that the LLM-provided renaming suggestions are similar to the original and representative of the class responsibilities, the prompt includes relevant context: the class type (e.g., *interface*, *enum*, etc.) and a list of methods and fields present in the class. The complete prompt is available in Appendix B.1.

Method renaming. Similarly, the transformation pipeline also filters methods. This includes those that have annotations (e.g., *@Bean* or *@Override*), getters and setters, and canonical method names (e.g., *hashCode* or *toString*). Furthermore, methods from interfaces that extend library code are excluded to prevent breaking external contracts. Also excluded are potential API entry points, i.e., public methods with no usage within the project. All other methods are included (e.g., behavioral or internal methods).

The context provided to the LLM for generating naming suggestions includes the method name, body, and containing class name. The complete prompt is available in Appendix B.2.

Variable renaming. The transformation applied to variables excludes public and protected fields, some annotated variables (e.g., *@Column*, to preserve database schema mappings), and *enum* constants, as they often have domain-specific values and are used externally, for example in database connections. Because renaming them externally is not feasible, these variables are excluded from the renaming process.

Unlike for class and method renaming, which process elements individually, variable renaming uses batched LLM requests to generate naming suggestions. All eligible variables from a single file are grouped into batched LLM calls to reduce API costs. The prompt still provides sufficient context for each variable, including type (e.g., *String, CustomObject*), location (e.g., *field, local variable, constant*), and containing scope (method or class name). The complete prompt is available in Appendix B.3.

3.1.2 ***Generating substitute names.*** Once all valid identifiers are collected, the pipeline relies on OpenAI's GPT-5 mini[4] to generate semantically similar names for the identifiers. The LLM is instructed to generate a series of suggestions, from most to least fitting, based on the old name and relevant context. The pipeline attempts to rename using each generated suggestion, checking for name collisions with other, already-existing, identifiers. If none of the suggestions are suitable, the identifier is skipped to keep the codebase compiling. On average, about 900 identifiers are renamed per instance, with only a few percent being skipped.

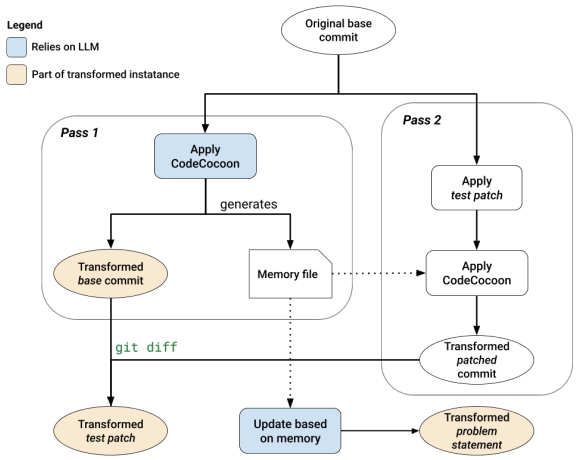

Figure 1: Memory-based two-pass system for creating the updated instance when applying renaming transformations.

3.1.3 ***Memory-based deterministic transformation.*** Each dataset instance contains fields that must be updated to create the transformed variants: *base commit, test patch, problem statement*, and *interface description*.

To ensure the base commit and test patch are using the same renames for all identifiers, CodeCocoon operates in two passes, as shown in Figure 1. The pipeline first applies the renaming transformations without memory, on the base commit. This means that an LLM-generated list of suggestions is created for each identifier. If the rename is successful, the new name is saved in a persistent memory file, where each identifier is mapped to the new name using a generated unique signature.

The second pass starts by applying the test patch on the original base commit and then re-applies the renaming transformations. In this phase, the new name for each identifier is collected from memory and updated directly, without prompting the LLM. This ensures a deterministic renaming process. We operate under the assumption that the test patch only changes test files, which holds for our chosen dataset.

Once the two CodeCocoon iterations are complete, the pipeline creates and stores the updated instance. The new base commit is obtained from the first CodeCocoon run, as explained above. The test patch is obtained by generating a *git diff* patch between the branch containing only the transformed base commit and the one having the test patch pre-applied. This two-pass approach ensures that the test patch in the transformed instance contains the correct names for any changed identifiers.

Because the problem statement and interface description may reference identifiers in the codebase, they must also be updated.

[3] https://github.com/JetBrains-Research/CodeCocoon-Plugin
[4] https://developers.openai.com/api/docs/models/gpt-5-mini

We decided to rely on a large language model to update them based on CodeCocoon's memory. Thus, an instance of GPT-5 mini is prompted to update all identifier references based on the memory file. The complete prompt is available in Appendix B.4.

3.2 Structural Transformations

In addition to identifier renaming, this study applies two types of structural transformations: changes to control flow and method ordering. We chose control flow transformations because of their widespread application in metamorphic testing of LLMs for code [5]. We selected method ordering because LLMs may rely on pattern recognition rather than understanding [28], especially when producing solutions for popular evaluation sets. Moreover, a recent study shows that LLMs for code are sensitive to the ordering of methods within the input [33]. Thus, we hypothesize that the agents would rely less on memorized patterns if the code, and therefore the token sequence, is ordered differently.

3.2.1 ***Control Flow Structural Transformations.*** The first type of structural transformation is based on the work of de Koning et al. [15]. Using the tool available in this work, our transformation pipeline applies the following transformations: For to While (F2W), Reverse If (RevIf), Nest Else If (NestEI), Swap Equals operands (SEO), Swap Relational operands (SRO), Expand Unary increment (EUI). These transformations are applied to all Java source files for each instance.

3.2.2 ***Method Ordering Transformation.*** A different type of transformation that changes the code, and therefore the LLM token sequence, is the reordering of methods within classes. The new order is chosen from a set of 10 random permutations. For each permutation, a displacement score is calculated, and the one with the highest displacement is selected. This displacement score maximizes the total distance methods move from their original positions, calculated as the sum of absolute index differences ($\sum_{i=1}^{n} |index(m_i) - i|$, where n is the total number of methods, m_i is the method at position i in the original order, and $index(m_i)$ is the position of method m_i in the reordered list).

3.3 Setup and Evaluation

3.3.1 ***Dataset.*** We use the SWE-rebench-V2 dataset [7], which contains containerized SWE tasks, enabling us to assess coding agents' capabilities by running related test suites. The tasks are sampled from two Java repositories and filtered for non-zero baseline success rate and benchmark-reported task solvability. We evaluate on 25 instances in total: 7 from the *googleapis/java-storage* project and 18 from the *eclipse-ee4j/yasson* project (instance IDs are listed in Appendix A.2). Alternative datasets, such as SWE-bench Multilingual [45] or DPAIA [17], were unsuitable due to low per-instance test counts or missing interface information. Without interface information, agents may produce correct solutions but use different identifiers for newly created classes and methods, leading to a compilation error and an inaccurate success score of 0.

For each base instance, three transformed instances are created: (1) all renaming transformations applied, (2) all renaming transformations plus the Control Flow Structural Transformation, and (3) all renaming and both structural transformations.

For renaming, instances are constructed by applying the corresponding transformations to both the base commit and its test-patched version. It is required to have the same renaming mappings on both versions to generate a correct transformed test patch, as explained in Section 3.1.3. For structural transformations, a one-pass application to the base commit is sufficient because the test patch does not change from these transformations.

3.3.2 ***Prediction & Evaluation.*** For prediction and evaluation, we rely on a modified pipeline based on an JetBrains internal evaluation harness, which is described below.

Autonomous coding agents use tools to read and modify the codebase. The pipeline relies on this functionality to generate and aggregate predictions. The pipeline uses a Docker container for each run, with the agent having access to the corresponding base commit. The agent is instructed to solve the task described in the *problem statement* by modifying the code. To prevent evaluation errors due to inconsistent naming of agent-created identifiers, *SWE-rebench-V2* provides additional information about the newly added *interfaces*, which is also given to the agent. The complete agent prompt is available in Appendix B.5. Once the agent completes the changes, the pipeline extracts them in a *.patch* file, which is later used for evaluation. Relevant metadata is also collected, including LLM token usage and cost.

To evaluate, the pipeline also prepares a separate Docker container for each instance. The repository is cloned, and the relevant base commit is checked out. The agent-generated patch is applied to the base commit, followed by the test patch from the dataset. All tests listed in the dataset instance are executed (this includes `FAIL_TO_PASS` and `PASS_TO_PASS`). The test pass-rate is recorded.

Because autonomous coding agents rely on large language models, they are inherently non-deterministic. We attempt to minimize the effects of this variance in agent behavior by generating and evaluating 10 predictions for each instance-transformation pair. Furthermore, all variance is reported, for example, by including the standard deviation in our results. For certain metrics, we rely on the median instead of the mean to reduce the impact of outliers.

3.3.3 ***Coding Agents.*** The goal of our study is to assess the robustness of coding agents; thus, we chose agents that rely on large language models that score highly on established benchmarks such as SWE-rebench [6] and SWE-bench-Verified [23]. The agents are accessed through a JetBrains internal LiteLLM gateway for inference.

Claude Code. Anthropic's Claude family of models score highly on code-related benchmarks [26], thus we chose their agent, Claude Code. Version 2.1.109 was used in the evaluation. This version relies on Claude Sonnet 4.6 and Claude Haiku 4.5 as underlying large language models. The agent has access to tools for reading, editing, and running code.

Codex. The study used the CLI variant of Codex, version 0.130.0, relying on GPT-5.1-codex as the model of choice. Similarly, Codex also has tools that can read, modify, and run the code. Codex CLI was running in *yolo* mode, allowing it to make edits and run commands without manual approval.

4 Results and Analysis

This section presents the findings on the robustness of Claude Code and Codex CLI to metamorphic transformations that change identifier names and code structure while preserving behavior. We present the results in relation to the research questions and analyze four variations of each instance:

- *Regular* – the untransformed baseline.
- *Renaming* – applies the identifier renaming of Section 3.1.
- *+ Control-flow* – adds the control-flow restructuring (from Section 3.2; includes *Renaming*).
- *+ Method-ordering* – adds method shuffling (from Section 3.2; includes *Renaming* and *Control-flow*).

Each agent is analyzed independently, based on 10 repeated runs for each instance-transformation pair. First, we conduct an *across-instances* analysis by aggregating the runs for each instance type and comparing them with the baseline using a *Wilcoxon signed-rank test*, reporting the p-value. The aggregation is performed using the mean of the success scores, since the scores are already normalized to 0-100. Throughout the text, we use the terms "success rate" and "success score" interchangeably. For cost and token metrics, we aggregate using the median to account for the very large variance in the data. We explore the variance further in Section 4.2.

Next, we run an equivalence analysis by means of a *two one-sided t-test (TOST)*. And lastly, we conduct a rank-based analysis using the *Friedman test* [18] with the *Nemenyi post-hoc test* [16]. The Friedman test is a non-parametric test that allows us to assess the significance of differences among the four variants, across repeated measurements (i.e., instances). The Nemenyi post-hoc test computes the average rank and allows us to compare two treatments by assessing whether the difference in average rank exceeds the critical distance (CD) [16].

Of the 1960 runs executed, 1878 (95.8%) completed the evaluation pipeline and are used for analysis; the remaining 82 are excluded because their evaluation pipeline did not finish successfully, e.g., due to a test patch application failure caused by a git conflict. Due to resource limitations, these runs were not repeated. Appendix A.3 shows a summary of the metrics for the success score, cost, and total tokens for each agent and each of the four variants of instances.

4.1 RQ1: Robustness of task success rate

4.1.1 ***Across-instance analysis.*** Across instances, the paired differences in success rate between the *Regular* baseline and each transformed variant are small for both agents, and no transformation produces a statistically significant change in task success (all Wilcoxon signed-rank p-values above 0.05). For Claude Code, the success rate trends positively (Δ is +2.6, +1.6, and +2.1 percentage points for the three transformed variants, respectively), while for Codex, it varies around zero (Δ is −0.3, +1.3, +0.1, respectively).

However, performance varies considerably even when the same task is run multiple times. Running an instance 10 times under the same conditions, the success score has an average run-to-run standard deviation of 3.6 percentage points for Claude Code and 10.6 percentage points for Codex. This is larger than the biggest difference measured between *Regular* and any transformed version (+2.6 points for Claude Code and +1.3 for Codex).

Table 1: Task success per agent and condition. *Zero-score %* is the ratio of runs that failed compilation (before or after the test patch application). *Cond. score* is the conditional mean success score, with standard deviation, across instances. It is computed only on runs that were compiled. Δ is the conditional mean success difference from *Regular*.

Agent	Instance type	Zero-score %	Cond. score	Δ
Claude Code	Regular	4.1%	93.56 ± 7.27	—
	Renaming	1.6%	93.58 ± 7.27	+0.02
	+ Control-flow	2.8%	93.51 ± 7.28	−0.05
	+ Method-ordering	2.4%	93.72 ± 7.36	+0.16
Codex	Regular	6.9%	94.12 ± 7.06	—
	Renaming	7.6%	93.83 ± 7.02	−0.29
	+ Control-flow	5.4%	93.93 ± 7.10	−0.19
	+ Method-ordering	6.2%	93.88 ± 7.12	−0.24

This high variance in the success rate mostly comes from runs that achieve a score of zero, for example, because the code provided by the agent does not compile. These types of errors are a genuine agent failure and should be taken into account, but they severely skew the mean success rate. We decided to report failures and non-zero runs separately. Table 1 reports, for each condition, both the ratio of zero-score runs and the mean success rate on the remaining non-zero runs, i.e., those that compile after the agent's prediction and test patch application. After filtering, the three transformed types are ±0.3 points from the baseline for both agents.

There are 78 runs in which the code did not compile during evaluation, resulting in a success score of 0. In 32 of them, the errors are due to syntax errors in the patched code, while the other 46 are due to incorrect names for added identifiers. All newly added identifiers are specified to the agent, so it should be able to name them accordingly. The failed runs are evenly distributed across conditions (18-21 zero-score runs per instance type, including baseline).

Finding 1.1: The differences in overall success rate come from a few total failures, where the agent's code does not compile. Conditional on compilation, the success rate remains within ± 0.16 percentage points of the baseline for Claude Code and decreases by at most 0.3 percentage points for Codex.

4.1.2 ***Equivalence test.*** A non-significant result based on the Wilcoxon test only means that no difference can be detected, not that the conditions are the same. Thus, we conducted an equivalence analysis using the *two one-sided t-tests* (TOST). It checks, for each instance, if the mean success score of the transformed variant is within a ±2.5 percentage points of the mean success score of the *Regular* instance. We use all runs for each condition, including those that score 0. We chose this margin because it is smaller than the run-to-run standard deviation of both agents. For all transformations, 19 of 25 instances are confirmed equivalent for Claude Code, and 13 of 24 for Codex. The rest are inconclusive rather than a significant difference. This is due to the agents' run-to-run variance and is more common with Codex, which has higher variance, thus having 11 inconclusive instances.

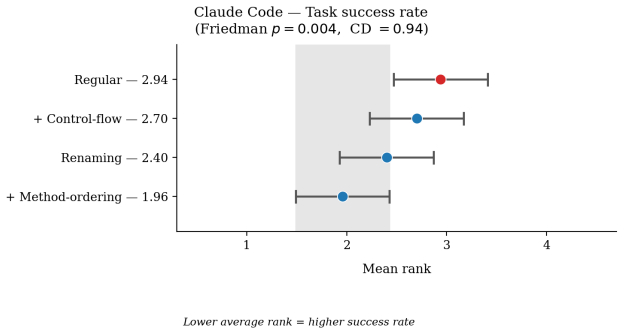

Figure 2: Results of statistical comparison based on the Friedman test with Nemenyi post-hoc test on task success rate for Claude Code. A red dot shows that a variant was significantly outranked by the top-ranked one. *Regular* is significantly lower than the top-ranked *+ Method-ordering*.

Finding 1.2: The instance-level equivalence test finds that metamorphic transformations do not have a general effect on either agent's success rate. 19 of 25 instances for Claude Code and 13 of 24 for Codex are within ±2.5 percentage points of the baseline. The remaining instances are inconclusive due to having larger run-to-run variance.

4.1.3 ***Rank-based analysis.*** The rank-based Friedman test across the four variants finds a significant change in Claude Code's task success ($p = 0.004$). The Nemenyi post-hoc (Figure 2) ranks *+ Method-ordering* highest. The single variant in which the average rank is at least the critical distance below the highest rank is the *Regular* baseline ($p = 0.037$). For Codex, none of the variants are outside the critical distance.

It is worth noting the direction of the difference. *Regular* has the lowest average rank, meaning that Claude Code achieves a higher success rate on the transformed versions than the baseline. We hypothesize that, due to changes in the code, the agent explores more rather than relying on patterns recognized from the training data. Because of this exploration, it has more relevant code in its recent context, thus reducing the failure rate.

Finding 1.3: For Claude Code, *+ Method-ordering* significantly outranks the *Regular* baseline in success rate. We hypothesize this is due to additional exploration of the codebase, which may bring relevant information into the model's context. The Friedman test does not find any significant difference for Codex.

4.1.4 ***Interpretation.*** Overall, both agents seem robust to the applied metamorphic transformations. The equivalence tests confirm that most instances are within 2.5 percentage points of the *Regular* baseline in success rate. Furthermore, the mean paired differences do *not* have significant p-values based on the Wilcoxon test. Filtering on non-zero success rates, the decrease in success for Codex may indicate a possible trend; however, there is insufficient data to reach a definitive conclusion.

Due to the non-deterministic nature of LLM-based coding agents, each run will have a different outcome. This makes it more difficult to extract true transformation effects from noise, especially for numeric metrics. We conducted the Friedman test, which only considers the ordering among variants and does not quantify differences. The results show a statistically significant difference between the *Regular* baseline and the *+ Method-ordering* variant, which contains all transformations. We interpret this as an indication that the other tests may not have sufficient data points to separate the run-to-run noise from a real effect and reach the significance threshold.

***RQ1* Answer: Robustness of task success rate**

Both agents are generally robust. The paired comparisons show no significant change in task success (all $p > 0.05$), and most instances are confirmed equivalent to *Regular* within ±2.5 points (19 of 25 for Claude Code, 13 of 24 for Codex). The rank-based Friedman test finds a significant difference between *Regular* and *+ Method-ordering* for Claude Code, with the transformed variant outranking the baseline. We hypothesize that this increase in Claude Code's success rate could be attributed to additional exploration of the codebase. There is no significant difference for Codex.

4.2 RQ2: Impact on resource and tool usage

It is important to preface this analysis with a discussion on the variance of the metrics (based on the data in Appendix A.3). Standard deviations are large across cost and token metrics, due to the many layers of non-determinism associated with LLM-based agents. The analysis is based on 10 repeated runs for each instance-transformation pair. Large standard deviations are also present in the baseline runs, so the effect is not strictly due to the transformations. There are two sources for this variance. First, the problems the agents are evaluated on (25 in total) vary in size and complexity. As an example, for Codex on the *Regular* variant, the per-instance median ranges from 0.2M to 6.2M tokens, across tasks. This is a variance we would expect.

However, there is a large amount of within-instance variance across re-runs under the same conditions. As an example, on instance 246 from *eclipse-ee4j/yasson*, on the *Regular*, baseline version, eight runs achieve an almost identical success score. Yet, the total number of used tokens ranges from 4.0M to 8.8M. The *+ Method-ordering* version of the same instance has total token usage varying from 1.7M to 19.1M, with all ten runs achieving the same success score. Here, the most expensive run uses more than 10 times as many tokens as the cheapest run while solving the same task.

One factor contributing to these variations is the number of API calls made in a given run. For Codex and Claude Code, every LLM API call adds the entire existing conversation as context, so each new tool call (e.g., those for reading or editing files) accumulates tokens disproportionately to the tool call. Both tools use shell commands at almost every step, and the output is then sent back to the underlying LLM, increasing token usage. A Pearson correlation analysis shows a clear relation between the number of commands and the total tokens used ($r = 0.97$). The same task can therefore use a very different number of tokens from one run to the next, due to the agent's tool choices, without affecting task success.

Table 2: Cost and token usage difference per agent and transformation type, shown as the mean paired difference relative to the *Regular* baseline. The Wilcoxon signed-rank p-value is shown in parentheses.

Agent	Transformation	Input tokens	Output tokens	Total tokens	Cost
Claude Code	Renaming	−1.1% (0.916)	−3.7% (0.113)	−3.7% (0.339)	−3.7% (0.113)
	+ Control-flow	+18.1% (0.063)	−3.4% (0.241)	−1.9% (0.312)	−1.8% (0.275)
	+ Method-ordering	−9.9% (0.596)	+1.1% (0.615)	+4.7% (0.525)	−0.2% (0.200)
Codex	Renaming	+0.1% (1.000)	−0.4% (0.855)	+5.8% (0.643)	+0.4% (0.922)
	+ Control-flow	+15.7% (0.089)	−6.2% (0.768)	+13.5% (0.208)	+6.9% (0.375)
	+ Method-ordering	−3.1% (0.406)	−2.4% (0.966)	+9.9% (0.584)	+1.8% (0.900)

Table 3: Tool usage per agent. The *Regular* column is the per-run average for the baseline. Each transformation column is the mean paired difference relative to *Regular*, with the Wilcoxon signed-rank p-value in parentheses.

Agent	Metric	Regular	Renaming	+ Control-flow	+ Method-ordering
Claude Code	Tool calls	17.2	−2.0% (0.756)	−3.0% (0.431)	−1.2% (0.787)
	Searches	2.9	+1.9% (0.417)	+6.3% (0.963)	+14.6% (0.175)
	Lines read	679	+7.7% (0.119)	+5.3% (0.378)	+15.7% (0.048)
Codex	Tool calls	61.5	+1.4% (0.410)	+3.6% (0.390)	+2.8% (0.623)
	Searches	18.5	+4.9% (0.290)	+7.0% (0.160)	+13.5% (0.121)
	Lines read★	1393	−0.3% (0.944)	+8.6% (0.188)	+14.3% (0.128)

★ Codex has no dedicated read tool. The number of lines read is approximated from shell commands.

4.2.1 ***Across-instance analysis.*** Table 2 shows for cost and input, output and total tokens, the mean paired difference relative to the *Regular* baseline, calculated as $\frac{\sum_{i=1}^{n}(\text{transf}_i - \text{reg}_i)}{\sum_{i=1}^{n}\text{reg}_i} \times 100$, where each reg_i / transf_i are the instance's median over its runs for the *Regular* and *Transformed* versions respectively. None of the metrics differ significantly from the baseline, with all p-values above 0.05.

Token usage. Looking at input tokens, which could be considered a proxy of how much the agent explores the codebase, the biggest difference is +18.1% for Claude Code with the *+ Control-flow* transformations. This is not significant ($p = 0.063$) and also unstable. Because the conditions are cumulative, *+ Method-ordering* also includes the control-flow restructuring, but the input tokens decrease by −9.9% compared to the baseline. If the control-flow transformations had a real effect, it should also be detectable when applying an additional transformation that reorders methods. For Codex, the same pattern appears: +15.7% for *+ Control-flow*, but −3.1% for *+ Method-ordering*.

Only small differences are visible for the output tokens, which is expected because the solution to the task should not depend on the codebase itself if the transformations make it semantically equivalent and do not change behavior.

For total tokens (which include input, output, and cached input tokens), the differences are small overall and non-significant for both agents. This total count is predominantly made of cached tokens, i.e., the context that is re-sent to the model at every API call (about 94% of the total for Claude Code and 91% for Codex). The total token count, therefore, reflects mostly re-sent context.

Cost. While cost is a metric calculated directly from token usage, since different types of tokens are priced at different levels, we are considering it separately as well. Cost is also a more relevant metric in the practical sense, compared to the number of tokens used. The cost remains within ±7% across any of the transformations, and the measured differences are small relative to the data's variance.

> **Finding 2.1:** Across instances, the transformations show no measurable effect on the agent's token usage or cost. All paired differences are small relative to the run-to-run variance, and non-significant, with cost staying within ±7% of the baseline.

Tool usage. Codex and Claude Code represent and report their tool usage and steps differently, but for both, we can extract the number of tool calls, searches, and lines read.

For Claude Code, usage is constructed from traces uploaded to Langfuse[5] at runtime. The paired comparisons for Claude Code are based on 21, 17, and 13 traced instances from the three transformed variants, respectively. The rest were unfortunately not captured, and resource limitations meant we could not run them again. However, these numbers are recorded only for the top-level agent. When it delegates to an *Explore* sub-agent (in about 80% of runs), that sub-agent's own searches and reads are marked as a single tool call. Thus, the numbers shown for Claude Code represent a lower bound. For Codex, each step is recorded in the prediction output logs, and all runs from the 24 used instances had complete telemetry data.

Table 3 shows the *Regular* per-run average and the mean paired difference for each transformation type for both models, with the Wilcoxon signed-rank p-value in parentheses.

Both agents make a similar number of tool calls across versions (within 3% of the baseline for Claude Code and 4% for Codex). There are trends of increased exploration, where both agents appear to search and read more under *+ Method-ordering*: +15.7%

[5] https://cloud.langfuse.com/

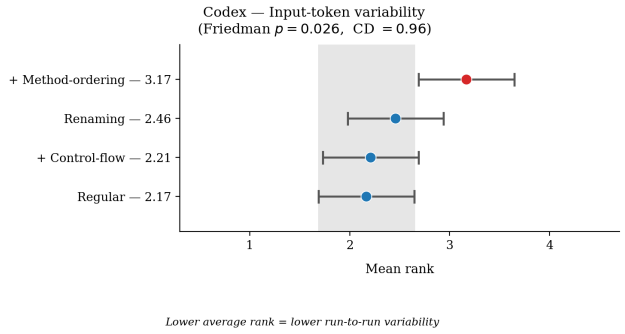

Figure 3: Results of statistical comparison based on the Friedman test with Nemenyi post-hoc test on the run-to-run input token variability for Codex. A red dot shows that a variant was significantly outranked by the top-ranked one. *+ Method-ordering* is significantly more variable than *Regular*.

for Claude Code ($p = 0.048$, but from only 13 instances for which traces were available) and +14.3% for Codex (not significant). The increase in searches is +13.5% for Codex and +14.6% for Claude Code (neither significant).

Finding 2.2: Both agents have a positive trend in searches and lines read as more transformations are added. While Claude Code's lines read metric technically reach significance (+15.7%, $p = 0.048$), this effect comes from only 13 traced instances, so it might not generalize. We therefore interpret the results as a possible tendency of both agents, rather than a clear effect.

4.2.2 ***Rank-based analysis.*** We applied the rank-based Friedman test across multiple metrics. While most did not have statistically significant results, the input token variability metric for Codex did. The test examined the run-to-run variance of the input tokens (using the relative Brown-Forsythe deviation [8] across each instance's 10 runs). The Friedman test finds a significant difference across the four variants ($p = 0.026$), and the Nemenyi post-hoc (Figure 3) ranks *Regular* highest, i.e., the most consistent. The single variant that is significantly outranked is *+ Method-ordering* ($p = 0.037$). Input tokens are a proxy of how much the agent explores the codebase. Since this looks at variability rather than the absolute number of tokens, it indicates that, in instances where all transformations are applied, Codex relies more on exploration in some runs. An alternative explanation could be that it needs to read more of the tools' output, for example, a full build error if it produces incorrect code. There are no significant results for Claude Code.

Finding 2.3: On the *+ Method-ordering* variant, Codex has a significantly larger run-to-run variance in input tokens ($p = 0.026$). This means its consumption of input tokens is less predictable on the same task when metamorphic transformations are applied. One explanation is that the transformations make relevant code harder to locate, so the agent explores more.

4.2.3 ***Interpretation.*** Overall, both agents seem mostly robust to the applied metamorphic transformations with respect to resource consumption. The paired comparisons show that cost and token metrics remain close to the *Regular* baseline, with the mean paired differences not significant according to the Wilcoxon test. However, the number of search commands and lines read increases with more transformations applied, suggesting a trend. Because of the limited number of data points, we cannot say with certainty that it holds in general; rather, we note it as an observation.

We do not report an equivalence test here, as we did for the success rate. The run-to-run variance is so large for cost and token metrics that choosing a meaningful margin is impossible.

The Friedman test, which looks at rankings rather than numerical values, can help identify trends. For resource consumption, the results show a statistically significant difference between the *Regular* baseline and *+ Method-ordering* for Codex in the input token variability. We interpret this as an indication that the transformations make the agent less predictable, as it may need to explore more of the codebase to find the relevant code.

***RQ2* Answer: Robustness of resources and tool usage**

Both agents show robustness. Cost stays within ±7% of the baseline and no token metric changes significantly. Yet, Codex has a significantly higher run-to-run variance in input token usage, which can serve as a proxy for how much the agent explores the codebase. This aligns with the positive trends we saw for both agents in the number of searches and lines read as more transformations are applied.

4.3 RQ3: Generalization of robustness trends

Based on the previous research questions, we conclude that Claude Code and Codex are largely robust to metamorphic transformations in both task success and resource consumption. Prior work on metamorphic testing of standalone LLMs for code reaches a different conclusion. In these works, large language models often produce code in a single pass, without access to any tools and with little to no feedback.

A study by de Koning et al. [15] looks at the robustness of program repair and applies similar transformations, renaming identifiers and restructuring code. They use two benchmarks (Defects4J and GitBug-Java) and report that all seven evaluated models perform worse on the transformed versions, with patch-generation success dropping between 4.1% (for GPT-4o) and 16.0% (for Llama-3.1). They attribute this to data leakage and memorization, demonstrated by a strong correlation between the performance drop and the negative log-likelihood of the original problems, which is a proxy for how familiar the model is with them.

A separate study by Orvalho and Kwiatkowska [28] evaluates nine LLMs on their output prediction capabilities. The LLMs are given a program and a concrete input and must state what the program returns, which requires them to reason about the code's behavior. The programs are taken from the LiveCodeBench and CruxEval datasets, and they apply five semantics-preserving mutations, such as renaming variables, swapping if-else branches, and loop unrolling. The mutations do not change what the program

computes, so the predicted output should stay the same, but accuracy on the mutated versions drops by up to 70% for Gemini 3 and 68% for GPT-5.2.

Although the tasks, models, and benchmarks differ between our study and these two, and we cannot make direct numerical comparisons, there are clear differences in trends. Two studies show that standalone LLMs are affected by metamorphic transformations, whereas our results indicate that agents are largely robust to such transformations. Both studies report a decrease in LLM success; in contrast, our study shows that no transformation produces a significant change in average behavior for either Claude Code or Codex, with the conditional success rate remaining within ±0.3 percentage points of the baseline. We even see a possible indication that metamorphic transformations might increase Claude Code's success rate, which we hypothesize is due to the agent exploring more. A likely reason for this difference is the way agents operate. They can explore the repository, run its code and tests, and read the actual output. So even if the underlying LLMs in agents have the robustness issues reported in previous studies, this feedback loop and access to tools appear to compensate, making the agents more robust.

We also consider resource consumption and tool usage, and find no general, significant effects for either agent. We identified a possible positive trend in the number of searches and lines read for both agents, as well as higher variability in the input tokens used by Codex. However, due to the limited dataset size, we classify them as indications rather than conclusive effects.

The clearest finding is that, if transformations have any effect, it is smaller than the agent's own non-deterministic variability. Running the same instance ten times under the same conditions affects the variability in success rates more than any difference we measured between the baseline and the transformed versions. In practice, it is very unlikely that a developer using a coding agent would notice any differences after renaming identifiers or refactoring the codebase. But this large variance finding should be extrapolated to agent evaluation generally. Because a single run can score far from the average, agent evaluations should consider doing repeated runs on the same task.

***RQ3* Answer: Generalization of robustness trends**

Standalone LLM sensitivity does not generalize to coding agents. Prior work finds that single-pass models lose accuracy when semantics-preserving transformations are applied, by 4.1 - 16.0% for program repair [15] and up to 70% for output prediction [28]. In contrast, both agents we study stay within ±0.3 percentage points of the baseline on task success (RQ1). Their resource consumption is likewise unaffected, remaining within a few percent of the baseline. Still, both agents tend to explore more of the transformed code (RQ2).

5 Discussion

5.1 The cost of evaluating robustness

Measuring robustness at this scale is expensive. To separate a small effect from large variance, we need many repeated runs across many instances and transformations, with multiple agents. In total, the agent inference for our runs costs around $1200, and took over 570 hours of compute time. Most of the time (about 425 hours) was spent on the evaluation, which consists of building the project and running the tests for each individual run. The cost does not include that of the LLM we used to generate the renaming suggestions, GPT-5 mini (Section 3.1), for which we have no usage metrics, nor the cost of many runs we attempted on other datasets, which we found were unsuitable. So these numbers are lower bounds on both time and cost.

Even with 10 runs per instance type, there is still substantial variance, and increasing the scale would not be feasible. It seems that brute-force repetition does not scale well for evaluating the robustness of coding agents. This raises the question of whether agent robustness should be evaluated this way at all. The practical difference is hard to see, and the statistical difference is hard to measure at a reasonable cost. It may be more useful to focus on testing parts of the agent, rather than on the average-case behavior of the whole system.

5.2 Limitations

Due to time and resource constraints, the renaming transformations are applied only to the files in the original solution patch and the neighbors up to two degrees away. Because it does not cover the entire repository, the agent may still read untransformed code and recognize it from its training data. Furthermore, we had to limit our analysis to 25 instances, primarily due to cost. We do acknowledge that running more instances may have given further insights.

We also applied the transformations cumulatively rather than one at a time. The *+ Control-flow* variant includes the renaming, and *+ Method-ordering* includes both the renaming and the control-flow restructuring. Because of this, we cannot fully isolate the effect of each transformation on its own and can only reason about them together. We chose this design primarily for cost reasons, since testing every transformation in isolation and in combinations would have significantly increased costs. An ablation study, in which each transformation is run alone and in every possible combination, may lead to further insights into how each transformation contributes to the observed effects.

5.3 Threats to validity

Internal validity. Two aspects might affect the results of our experiments. First, the runs were collected in several batches and with two execution environments, a local and a remote one, so small differences between batches cannot be ruled out. We attempt to mitigate this by running predictions and evaluations in Docker containers, which should result in consistent runtime environments.

Second, our agents run on newer, stronger models than the standalone LLM studies we compare against [15, 28], so part of the robustness gap could come from base-model strength. Yet, agents operate in a fundamentally different way and rely on multiple steps to reach their result. So even if the same underlying models were used, a direct comparison would not be fair.

External validity. The study covers only two agents and 25 instances from two Java repositories, so the results may not generalize to other languages, repositories, or agents. The AI field also advances

rapidly, with more capable models released every few months, so findings on specific model versions may not hold in the future.

Construct validity. Task success is measured through the project's own test suites. Passing them is a proxy for a correct solution, but it does not rule out regressions that the tests miss. However, in most instances we analyzed, each has between 100 and 400 tests, which should cover most of the required functionality and limit the risk of missed regressions.

Lastly, the agents rely on proprietary models, meaning we do not have access to their internal reasoning traces. As a result, our findings are based only on the data and outputs we can directly observe. While it is a limitation of the study, these specific agents are used in practice for software development, and engineers interact with them in the same manner, as a black box. We therefore consider our results and insights highly relevant from a practical perspective.

6 Conclusions and Future Work

Non-deterministic large language models that are sensitive to small input changes, which do not alter program behavior. This work asked whether that weakness also appears in autonomous coding agents that rely on LLMs to work on tasks across repositories. We built *CodeCocoon*, a tool that applies metamorphic transformations to Java projects, and used it to rename identifiers, restructure control flow, and reorder methods. We ran Claude Code and Codex on 25 instances from the *SWE-rebench-V2* dataset, comparing the transformed versions to the originals across repeated runs.

Our results show that agents are generally robust to these metamorphic transformations. Still, we did find trends indicating that both agents explore more of the codebase, reading more lines and running more searches. For Codex, we also observed higher run-to-run variance in the version with all transformations applied, indicating the agent becomes less predictable. We interpret these results as evidence that the agents rely less on recognizing memorized patterns and more on discovering the codebase when transformations are applied. Standalone models lose accuracy under the same kinds of transformations, so the weakness observed at the model level appears to be mitigated once the model is placed in an agentic harness. A likely reason is the feedback loop: agents read and edit the code, run tests, and react based on the output, which allows them to fix any mistakes that might be introduced by LLM memorization.

This robustness is a practical observation, not a proof. The run-to-run variance of the agents is large, often larger than any measured effect the transformations might have. Measuring this kind of robustness is also expensive. We executed 10 runs per instance type, and this costs around $1200 for our 25 instances across two agents. Still, the variance is large, making it difficult to measure any effects. However, our results suggest that, in practice, a developer is unlikely to see an agent perform worse due to renamed identifiers or reordered methods.

In terms of future work, the clearest direction is finding cheaper ways to evaluate agent robustness. One possibility is predictive metamorphic testing, where a classifier predicts, from a set of features, which instances are likely to yield differences, so that the expensive agentic runs are performed only on those. Another way is to test parts of the agent rather than the entire agent, such as the underlying model's thinking traces or the repository exploration phase, which would also reveal where the robustness comes from. A third idea is adversarial transformations. Our method-ordering transformation selects the permutation that maximizes the displacement score across 10 random orderings. However, a search-based alternative may identify the ordering that most affects an agent and determine a worst-case performance bound. Furthermore, our renaming suggestions are generated to resemble the originals and mimic human-generated ones. An approach in which new identifier names are generated to deliberately confuse the agent might have a greater effect.

We close by answering the question in our title: *Are AI Agents for Code Robust to Metamorphic Transformations?* In practice, mostly.

Data Availability

The version of the tool used to apply the renaming and method-ordering transformations is available at: https://github.com/JetBrains-Research/CodeCocoon-Plugin/tree/adragoi/evaluation.
The replication package with raw results and analysis scripts is available at: https://github.com/dragoi75/CodeCocoon-EvaluationResults.

Acknowledgments

This work was conducted as an internship part of the AI for Software Engineering (AI4SE) collaboration between JetBrains and Delft University of Technology. We gratefully acknowledge the financial support provided by JetBrains, which made this research possible.

References

[1] Ali Al-Kaswan, Maliheh Izadi, and Arie Van Deursen. 2024. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[2] Anthropic. 2025. Claude Code. https://code.claude.com/docs/en/overview Agentic coding tool that runs in the terminal. Source repository: https://github.com/anthropics/claude-code.

[3] Leonhard Applis, Annibale Panichella, and Ruben Marang. 2023. Searching for quality: Genetic algorithms and metamorphic testing for software engineering ml. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1490–1498.

[4] Leonhard Applis, Annibale Panichella, and Arie van Deursen. 2021. Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1377–1381. doi:10.1109/ASE51524.2021.9678706

[5] Ali Asgari, Milan de Koning, Pouria Derakhshanfar, and Annibale Panichella. 2025. Metamorphic Testing of Deep Code Models: A Systematic Literature Review. *arXiv preprint arXiv:2507.22610* (2025).

[6] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2026. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents.

[7] Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. 2026. SWE-rebench V2: Language-Agnostic SWE Task Collection at Scale. arXiv:2602.23866 [cs.SE] https://arxiv.org/abs/2602.23866

[8] Morton B Brown and Alan B Forsythe. 1974. Robust tests for the equality of variances. *Journal of the American statistical association* 69, 346 (1974), 364–367.

[9] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2023. Quantifying Memorization Across Neural Language Models. arXiv:2202.07646 [cs.LG] https://arxiv.org/abs/2202.07646

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex

Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[12] Tsong Yueh Chen, Shing Chi Cheung, and Siu Ming Yiu. 1998. Metamorphic Testing: A New Approach for Generating Next Test Cases. *Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology* (1998). Reprinted as arXiv:2002.12543.

[13] Zhenpeng Chen, Jie M Zhang, Max Hort, Mark Harman, and Federica Sarro. 2024. Fairness testing: A comprehensive survey and analysis of trends. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–59.

[14] Shamse Tasnim Cynthia, Joy Krishan Das, and Banani Roy. 2026. Are We All Using Agents the Same Way? An Empirical Study of Core and Peripheral Developers Use of Coding Agents. *arXiv preprint arXiv:2601.20106* (2026).

[15] Milan de Koning, Ali Asgari, Pouria Derakhshanfar, and Annibale Panichella. 2026. A Metamorphic Testing Approach to Diagnosing Memorization in LLM-Based Program Repair. arXiv:2604.21579 [cs.SE] https://arxiv.org/abs/2604.21579

[16] Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research* 7, Jan (2006), 1–30.

[17] DPAIA. 2025. https://dpaia.dev/. Developer Productivity AI Arena.

[18] Milton Friedman. 1940. A comparison of alternative tests of significance for the problem of m rankings. *The annals of mathematical statistics* 11, 1 (1940), 86–92.

[19] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065* (2024).

[20] Hao Guan, Lingyue Fu, Shao Zhang, Yaoming Zhu, Kangning Zhang, Lin Qiu, Xunliang Cai, Xuezhi Cao, Weiwen Liu, Weinan Zhang, et al. 2026. SWE-Cycle: Benchmarking Code Agents across the Complete Issue Resolution Cycle. *arXiv preprint arXiv:2605.13139* (2026).

[21] Ruanqianqian Huang, Avery Reyna, Sorin Lerner, Haijun Xia, and Brian Hempel. 2025. Professional Software Developers Don't Vibe, They Control: AI Agent Use for Coding in 2025. *arXiv preprint arXiv:2512.14012* (2025).

[22] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *The Thirteenth International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2403.07974

[23] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[24] Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. 2025. The SWE-Bench Illusion: When State-of-the-Art LLMs Remember Instead of Reason. *arXiv preprint arXiv:2506.12286* (2025).

[25] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. *arXiv preprint arXiv:2409.02977* (2024).

[26] Spencer Mateega, Jeff Yang, Tiana Costello, Shaurya Jadhav, Nicole Tian, and Agustin Garcinuño. 2026. IDE-Bench: Evaluating Large Language Models as IDE Agents on Real-World Software Engineering Tasks. *arXiv preprint arXiv:2601.20886* (2026).

[27] OpenAI. 2025. Codex CLI. https://developers.openai.com/codex/cli Coding agent that runs in the terminal. Source repository: https://github.com/openai/codex.

[28] Pedro Orvalho and Marta Kwiatkowska. 2026. Are Large Language Models Robust in Understanding Code Against Semantics-Preserving Mutations? arXiv:2505.10443 [cs.SE] https://arxiv.org/abs/2505.10443

[29] Sotiris Pelekis, Thanos Koutroubas, Afroditi Blika, Anastasis Berdelis, Evangelos Karakolis, Christos Ntanos, Evangelos Spiliotis, and Dimitris Askounis. 2025. Adversarial machine learning: a review of methods, tools, and critical industry sectors. *Artificial Intelligence Review* 58, 8 (2025). doi:10.1007/s10462-025-11147-4

[30] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).

[31] Fazle Rabbi and Jinqiu Yang. 2026. HEJ-Robust: A Robustness Benchmark for LLM-Based Automated Program Repair. *arXiv preprint arXiv:2605.02215* (2026).

[32] Md Rafiqul Islam Rabin, Nghi D Q Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552. doi:10.1016/j.infsof.2021.106552

[33] Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. Order Matters! An Empirical Study on Large Language Models' Input Order Bias in Software Fault Localization. *arXiv preprint arXiv:2412.18750* (2024).

[34] Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 14116–14137.

[35] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. 2024. TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 30–34. doi:10.1145/3639478.3640024

[36] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[37] Sergio Segura, Gordon Fraser, Ana B Sánchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824. doi:10.1109/TSE.2016.2532875

[38] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. 23–30. doi:10.1109/APR59189.2023.00012

[39] Claudio Spiess, Prem Devanbu, and Earl T Barr. 2026. How Robustly do LLMs Understand Execution Semantics? *arXiv preprint arXiv:2604.16320* (2026).

[40] Stack Overflow. 2025. *2025 Stack Overflow AI Survey*. Stack Overflow. https://survey.stackoverflow.co/2025/ai#ai-agents-ai-agents

[41] Jianxun Wang and Yixiang Chen. 2023. A Review on Code Generation with LLMs: Application and Evaluation. *Proceedings - 2023 1st IEEE International Conference on Medical Artificial Intelligence, MedAI 2023* (2023), 284 – 289. doi:10.1109/MedAI59581.2023.00044

[42] Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 13818–13843.

[43] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*. ACM, 819–831. doi:10.1145/3650212.3680323

[44] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 37. https://arxiv.org/abs/2405.15793

[45] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. SWE-smith: Scaling Data for Software Engineering Agents. arXiv:2504.21798 [cs.SE] https://arxiv.org/abs/2504.21798

[46] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial Examples for Models of Code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. doi:10.1145/3428230

[47] Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. 2025. Survey on evaluation of llm-based agents. *arXiv preprint arXiv:2503.16416* (2025).

[48] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2020), 1–36.

[49] Jingming Zhuo, Songyang Zhang, Xinyu Fang, Haodong Duan, Dahua Lin, and Kai Chen. 2024. ProSA: Assessing and understanding the prompt sensitivity of LLMs. *arXiv preprint arXiv:2410.12405* (2024).

A Supplementary Materials

A.1 Study pipeline

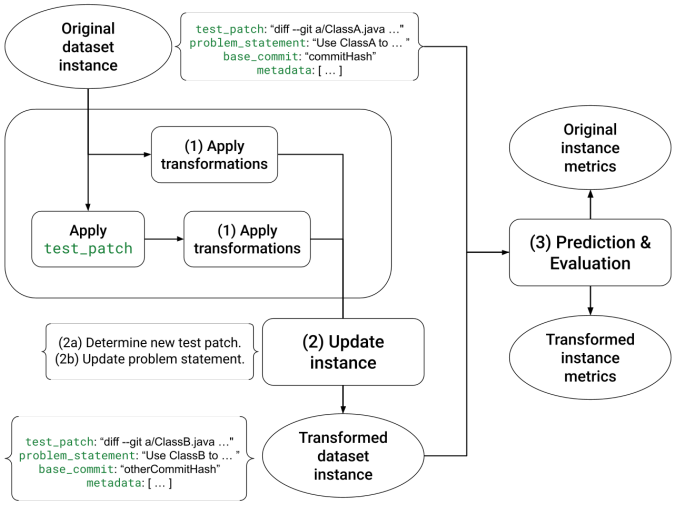

Figure 4: Study pipeline consisting of (1) Applying the metamorphic transformations, (2) Creating the transformed dataset, and (3) Prediction & Evaluation.

A.2 Subset of used instances

Table 4: List of SWE-rebench-V2 [7] task IDs, per project, used in the evaluation of our study.

Repository	Count	Instance IDs
googleapis/java-storage	7	102, 218*, 229, 308, 438, 677, 746
eclipse-ee4j/yasson	18	236, 246, 250, 264, 267, 286, 287, 288, 300, 395, 405, 493, 510, 546, 562, 576, 590, 675
Total	**25**	

* Only used for Claude Code

A.3 Overview of metrics

Table 5: Summary per agent and instance variant: *Regular* (baseline), *Renaming*, Renaming *+ Control-flow*, and Renaming + Control-flow *+ Method-ordering*. All values are mean ± standard deviation: for success rate across instance means; for cost and total tokens, across instance medians.

Agent	Condition	Instances	Success	Cost ($)	Tokens
Claude Code	Regular	25	89.67 ± 11.33	0.533 ± 0.309	1.20 ± 0.73M
	Renaming	25	92.28 ± 11.36	0.513 ± 0.326	1.16 ± 0.71M
	+ Control-flow	25	91.25 ± 15.52	0.523 ± 0.334	1.18 ± 0.84M
	+ Method-ordering	25	91.78 ± 14.16	0.532 ± 0.358	1.26 ± 0.86M
Codex	Regular	24	87.55 ± 16.42	0.637 ± 0.492	1.95 ± 1.72M
	Renaming	24	87.23 ± 18.09	0.635 ± 0.490	2.04 ± 1.81M
	+ Control-flow	24	88.80 ± 19.02	0.669 ± 0.500	2.16 ± 1.83M
	+ Method-ordering	24	87.68 ± 17.91	0.646 ± 0.452	2.12 ± 1.68M

B Prompts

B.1 Class rename suggestions prompt

Context

The context variables are resolved dynamically during execution:

- ⟨*class_type*⟩: The specific type of the Java entity, such as `interface`, `enum class`, `abstract class`, or a standard `class`.
- ⟨*class_name*⟩: The current name of the class.
- ⟨*methods_context*⟩: A list of method names within the class (e.g., *"The methods in the class are: [m1, m2]"*). This is omitted if the class has no methods.
- ⟨*fields_context*⟩: A list of field names within the class. This is omitted if no fields are present.
- ⟨*count*⟩: The number of alternative names the model is requested to generate.

System Prompt

You are an agent that proposes semantically similar Java class names. Your output is used in a metamorphic transformation pipeline. Your output will be parsed into JSON; strictly follow the required structure.

User Prompt

The name of the ⟨*class_type*⟩ is: ⟨*class_name*⟩
⟨*methods_context*⟩
⟨*fields_context*⟩
Return a JSON object with field 'suggestions' which is an ordered array of ⟨*count*⟩ Java identifiers, from most to least fitting.
Example structure:
`{"suggestions": ["BestFittingRename", "SecondBestFittingRename", ... ]}`
Every suggestion must be a valid Java identifier and semantically similar to the original name.

B.2 Method rename suggestions prompt

Context

The context variables are resolved dynamically during execution:

- ⟨*method_name*⟩: The current name of the method.
- ⟨*method_body*⟩: Entire text of the method's body.
- ⟨*class_name*⟩: The name of the class that contains the method.
- ⟨*count*⟩: The number of alternative method names the model is requested to generate.

System Prompt

You are an agent that proposes semantically similar Java method names. Your output is used in a metamorphic transformation pipeline. Your output will be parsed into JSON; strictly follow the required structure.

User Prompt

The current method name is: ⟨*method_name*⟩
The method body is: ⟨*method_body*⟩
The containing class name is: ⟨*class_name*⟩
Return a JSON object with field 'suggestions' which is an ordered array of ⟨*count*⟩ Java identifiers, from most to least fitting.
Every suggestion must be a valid Java identifier and semantically similar to the original name.

B.3 Variable rename suggestions prompt

Context

The following are resolved dynamically for each variable:

- ⟨*original_name*⟩: The current name of the variable.
- ⟨*declared_type*⟩: The variable's type (e.g., `String`, `List`, etc.).
- ⟨*scope_context*⟩: The location and scope of the variable (e.g., *"Local variable in class 'Example'"*).
- ⟨*naming_convention*⟩: The required casing format (`camelCase` or `UPPER_SNAKE_CASE`).
- ⟨*count*⟩: Number of suggestions requested per variable.

System Prompt

You are an agent used for refactoring Java code for metamorphic testing. You specialize in generating semantically similar variable names. Your output will be parsed into JSON; strictly follow the required structure.

User Prompt

Generate ⟨*count*⟩ semantically similar name suggestions for each of the following variables:
[For each variable:]
Variable: ⟨*original_name*⟩
Declared Type: ⟨*declared_type*⟩
Context: ⟨*scope_context*⟩
Required Format: ⟨*naming_convention*⟩

Return a JSON object with field 'renamings' which is an array of objects. Each object must have 'originalName' (the current variable name) and 'suggestions' (an array of ⟨*count*⟩ valid Java identifiers).
Example schema:
`{"renamings": [{"originalName": "old1", "suggestions": ["new1", "new2"]}]}`

Every suggestion must be semantically similar to the original name and follow the specified naming convention.
IMPORTANT: The 'originalName' field must exactly match the variable name provided above.
IMPORTANT: Refrain from using the old variable name in the new name (e.g., do NOT propose `newOwnerId` for `ownerId`).

B.4 Problem statement update prompt

Context

The context variables are resolved dynamically during execution:

- ⟨*problem_statement*⟩: The original problem statement text that requires updating.
- ⟨*interface_description*⟩: The original interface description text.
- ⟨*rename_mappings*⟩: CodeCocoon memory content (a list of mappings from old signatures to new names).

System Prompt

You are a technical documentation assistant helping to update code descriptions after refactoring transformations have been applied.
You will be given:

(1) An original problem statement
(2) An original interface description
(3) A mapping of old class/method names to new names

Your task:

- Update the problem statement and interface description to use the NEW names
- Keep the meaning and structure exactly the same
- Only change the class, method, and variable names according to the mapping
- Preserve all formatting, punctuation, and sentence structure
- If a name doesn't appear in the mapping, leave it unchanged

Important:

- Do NOT add new information
- Do NOT remove information
- Do NOT rephrase or rewrite the content
- ONLY update the names that appear in the rename mapping

User Prompt

Original Problem Statement
⟨*problem_statement*⟩

Original Interface Description
⟨*interface_description*⟩

Rename Mapping (OldName -> NewName)
⟨*rename_mappings*⟩

Now, update the problem statement and interface description with the new names.

B.5 Agent prediction prompt

Context

The context variables are resolved dynamically during execution:

- ⟨*problem_statement*⟩: The specific bug, feature request, or issue description provided for the repair task.
- ⟨*interface_info*⟩: Metadata regarding interfaces added or modified, used to ensure consistent naming and references within the implementation.

Prompt

Task: Fix the issue described below by making code changes.

Problem Statement:
⟨*problem_statement*⟩

Interface Information:
The following describes the interfaces that were added or modified in this problem. Use this information to properly name and reference these interfaces in your solution:
⟨*interface_info*⟩

IMPORTANT: Please do NOT add any tests. If you want to write tests to verify your implementation, please delete the newly added tests as the last step before submitting your solution.

C Fixing renaming errors

Due to indexing issues in some repositories containing multiple sub-projects, the usages of a small subset of methods are not consistently renamed, leading to compilation errors. To avoid discarding the entire instance, we apply a manual repair step. This consists of prompting Claude Code to repair the compilation errors generated by running `mvn clean test`. Claude Code is given access to the memory file generated by CodeCocoon, containing a mapping from the fully-qualified name, to the new identifier name. Once the fix is generated, the patch is human-reviewed to ensure only identifier-renaming changes are made and that there are no more compilation errors. The complete prompt used with Claude Code is presented below.

Claude Code Prompt

I have conducted a refactoring containing ONLY renaming of identifiers. However, some renamings have not completed successfully, so the code throws errors when running `mvn clean test`.
You have access to a mapping of the old to new names at `@<path/to/codecocoon/memory>`.

Please fix the errors by only making renaming actions (preferably rename usages that were missed, or rollback to the original name if necessary, e.g. because of inheritance).

D Use of Generative AI Tools

We used several generative AI tools during this thesis. Claude Code assisted in writing and debugging parts of the CodeCocoon codebase and the scripts used to aggregate the results and produce figures. Gemini helped brainstorming text structure and with finding papers for the literature search, and Grammarly was used to improve the writing style and clarity. These tools supported the work, did not replace it. The research ideas, study design, analysis, findings and conclusions are our own, and we verified the output of each tool before using it.