

Error Analysis of Automated Code Refactoring On Beginner-Friendly Codes

Piper Jeffries

Carson Kempf

Mia Mohammmd Imran

University of Missouri Science and Technology

Rolla, Missouri, USA

Abstract

Code refactoring is a crucial practice for beginner software engineers, as it improves code readability and structure without altering the behavior of the program. This study analyzes the behavior of an automated refactoring pipeline applied to beginner-oriented Python programs. The system combines complexity scoring, Abstract Syntax Trees, and a prompt-tree-driven strategy selector within a structured LangGraph workflow to generate refactored code while preserving program semantics. Using a benchmark of 974 programs, we evaluate refactoring outcomes across categories including computational tasks, data operations, strings, control flow, and introductory algorithms. Results show that 603 programs were successfully refactored, while 371 failures concentrated in a small set of recurring patterns, dominated by function-signature mismatches and subtle logical deviations. We introduce an error taxonomy grounded in these observations and assess complexity changes before and after transformation. The analysis highlights both the strengths and systematic limitations of LLM-based refactoring and identifies targeted opportunities for improving prompt design, routing heuristics, and validation mechanisms.

ACM Reference Format:

Piper Jeffries, Carson Kempf, and Mia Mohammmd Imran. 2025. Error Analysis of Automated Code Refactoring On Beginner-Friendly Codes. In . ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 Introduction

Refactoring restructures code without changing its external behavior, allowing programmers to produce clearer and more maintainable implementations. For beginners, this practice is especially valuable because it strengthens understanding of program structure while reducing unnecessary complexity. Automated refactoring tools can support this process, but they also introduce risks when the transformations applied by large language models do not preserve the original logic.

This work examines a multi-stage refactoring system designed to operate on beginner-friendly Python programs. The system integrates code normalization, syntax verification, Abstract Syntax

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, Washington, DC, USA

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

Tree construction, and complexity scoring. These outputs guide a decision process that selects an appropriate refactoring strategy through a prompt-tree classifier and executes it within a controlled LangGraph workflow. The design aims to ensure consistent, behavior-preserving transformations while preventing unnecessary modifications to simple or structurally stable programs.

To evaluate the reliability of this approach, we apply the system to 974 programs drawn from common introductory categories, including computational tasks, data and array operations, string processing, control flow, and basic algorithms. The results demonstrate both the strengths and limitations of automated refactoring: while many programs are successfully transformed, a substantial number fail due to recurring issues such as function-signature changes, incorrect logic adjustments, or formatting deviations. These patterns motivate the development of a structured error taxonomy that characterizes the dominant failure modes observed across the evaluation.

This study provides a detailed analysis of how automated refactoring behaves on novice-level code and identifies specific areas where constraints, prompts, and workflow rules can be refined to improve correctness and consistency.

2 Background and Key Concepts

In order to produce the best and most accurate code refactoring results, this project uses several foundational tools to analyze code and classify its structure in order to generate high-quality refactoring instructions.

These tools include:

- (1) Complexity Calculator
- (2) Abstract Syntax Tree (AST)
- (3) Prompt Tree
- (4) LangGraph Workflow

2.1 Complexity Calculator

The complexity calculator provides quantitative metrics, based on a set of predefined rules, that guide refactoring decisions. By measuring how difficult code is to understand, it identifies sections that are likely to confuse beginners. These measurements enable the tool to select the most appropriate refactoring strategy for each situation and establish clear thresholds for when refactoring is beneficial, preventing unnecessary modifications to code that is already maintainable.

2.2 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a structural representation of source code that expresses the logical relationships between language constructs. Instead of treating the code as raw text, an AST organizes the program into a tree of nodes, where each node corresponds to a meaningful syntactic element. For example, function, loop, conditional, assignment, or expression. This makes it possible to analyze code in a way that is robust, structured, and independent of formatting.

We chose to implement an AST because raw text analysis is too fragile for determining code complexity or identifying refactoring opportunities. For example, counting how many times the string "for" appears in a code does not reliably indicate how many actual loops exist. AST analysis provides reliable structure detection, language-aware complexity extraction, and a stable foundation for future refactoring. This allows our system to compute complexity features accurately and consistently by capturing the true logical form of the program.

2.3 Prompt Tree

A prompt tree is a structured decision system that selects the most suitable LLM prompt based on the characteristics of the input code. Instead of using a single generic prompt for all situations, the prompt tree analyzes the code complexity and structural features and chooses a specialized prompt that maximizes refactoring accuracy. This reduces the chance of using an overly strong or overly weak refactoring instruction, ensuring that the LLM receives a prompt tailored to the code's actual structure. Effectively improving consistency, reducing model hallucination, and ensuring the refactored output stays aligned with the original logic.

2.4 LangGraph

LangGraph is a workflow orchestration framework that manages the flow of information between the system's components. It ensures that each step, from code analysis to prompt selection to automated refactoring, follows a controlled, repeatable, and structured process.

3 Methodology

This section describes the full pipeline used in our refactoring system, including feature extraction, complexity scoring, code classification, prompt selection, and workflow orchestration through LangGraph.

3.1 System Overview

Our system consists of five sequential processing nodes:

- (1) Ingest & Baseline
 - (2) Routing
 - (3) Strategy Prediction & Prompt Refinement
 - (4) Targeted LLM Execution
 - (5) Measure & Learn

LangGraph is used to manage the order, transitions, and conditional routing between these nodes. Figure 1 shows the overview of the procedure.

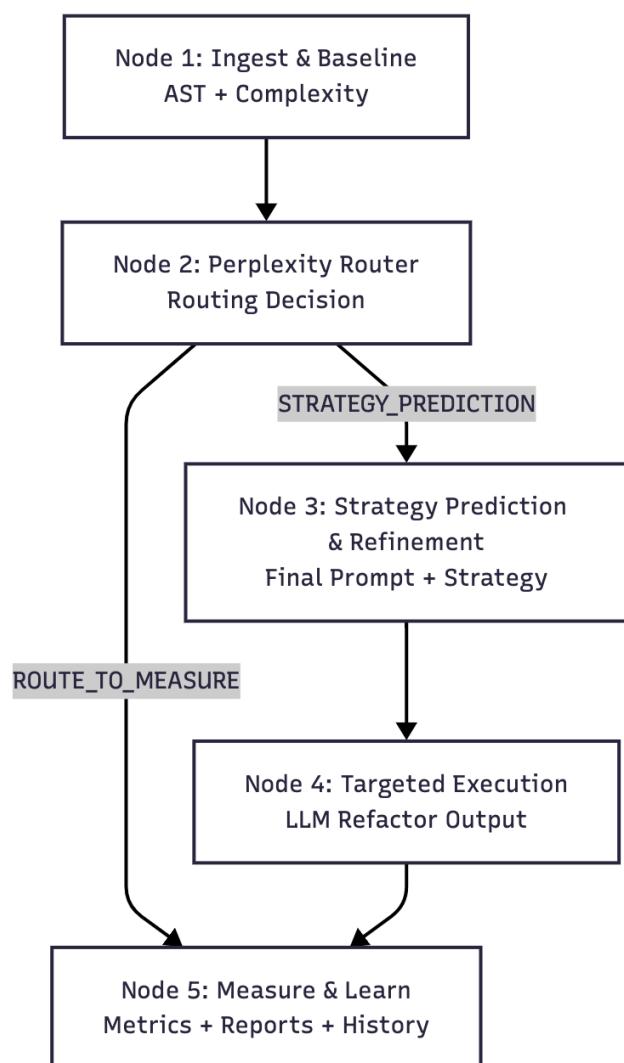


Figure 1: System Overview Workflow

3.2 Node 1: Ingest & Baseline (AST + Complexity Analysis)

This node receives the raw user input, which may include natural language description, source code, or both.

This stage performs:

3.2.1 Code Normalization.

- (1) normalize whitespace, line endings, and UTF-8 encoding
 - (2) remove natural-language text surrounding the code
 - (3) detect programming language
 - (4) sanitize malformed indentation when possible

3.2.2 Syntax Verification and AST Construction. The system attempts to parse the input into an Abstract Syntax Tree (AST). If parsing fails, the node performs automatic sanitation and retries. A

233 successful parse produces the baseline structural representation of
 234 the code.

235 **3.2.3 Complexity Metric Extraction.** The input code is
 236 analyzed using static analysis tools (specifically the Radon library) to
 237 extract quantitative metrics. Instead of relying on simple keyword
 238 counting, this node computes structural and physical attributes of
 239 the code:

- 241 • **Cyclomatic Complexity (CC):** A measure of the number
 242 of linearly independent paths through the program's source
 243 code.
- 244 • **Lines of Code (LOC):** The total physical lines of code,
 245 serving as a proxy for program size.

246 These features are then passed to the complexity calculator to
 247 produce a unified score.

248 **3.2.4 Code Complexity Rules.** In order to rigorously quantify
 249 code maintainability, we utilize a composite **Code Complexity**
 250 **Score.** This score combines the structural complexity (logic) with
 251 the physical size (verbosity).

- 253 • **1. Cyclomatic Complexity Definition** The complexity
 254 M is calculated using McCabe's formula, which analyzes
 255 the Control Flow Graph (CFG) derived from the AST:

$$256 M = E - N + 2P \quad (1)$$

- 258 • **2. Final Scoring Formula** We combine the structural metric
 259 (*CC*) with a weighted physical metric (*LOC*) to determine
 260 the final score:

$$261 Score_{complexity} = CC + (0.2 \times LOC) \quad (2)$$

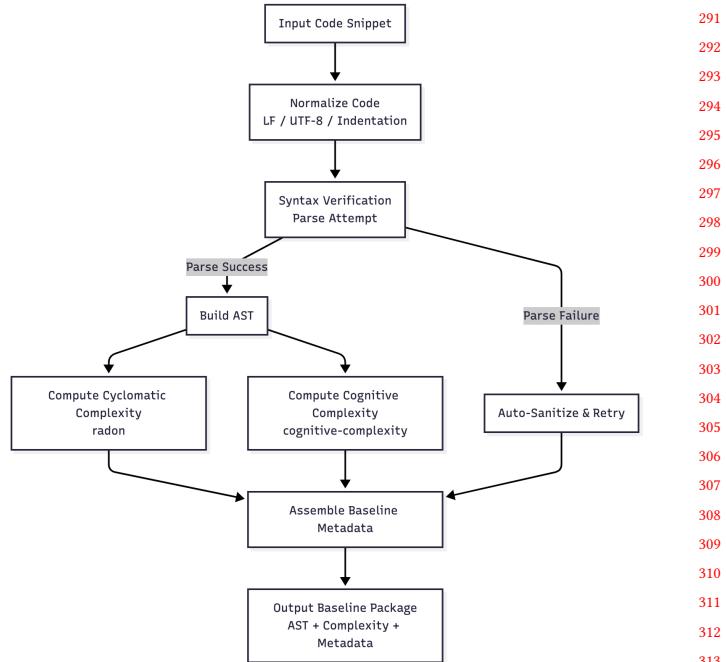
263 Note: We apply a weight of 0.2 to *LOC* to ensure that logic
 264 complexity weighs more heavily than simple length.

265 **3.2.5 Note About Control Flow Graph Representation Used**
 266 **in Cyclomatic Complexity Calculation.** In the context of our
 267 Python analysis, the Control Flow Graph represents the program's
 268 logic topology.

- 269 • **Nodes (N):** Represent "basic blocks" of code—sequences
 270 of instructions (assignments, function calls) that execute
 271 linearly without branching.
- 272 • **Edges (E):** Represent control transfer. A simple assignment
 273 moves to the next line (1 edge), whereas a conditional state-
 274 ment like *if* or *for* generates multiple outgoing edges
 275 (branching paths).
- 276 • **Calculation:** Our system uses the *radon* library to con-
 277 struct this graph in memory. It then counts the number of
 278 linearly independent paths through this graph to derive the
 279 Cyclomatic Complexity score.

281 3.2.6 How It's Used in Refactoring.

- 282 (1) Programmer inputs a piece of code.
- 283 (2) The system calculates the *CC* and *LOC*.
- 284 (3) A final *Complexity Score* is produced.
- 285 (4) If score is $\geq 6.0 \rightarrow$ Code is classified as **Complex**. The
 286 system triggers a full refactoring workflow.
- 287 (5) If score is $< 6.0 \rightarrow$ Code is classified as **Simple**. The system
 288 triggers a "Readability Improvement" workflow (e.g., better
 289 variable names, comments) rather than structural changes.



314 **Figure 2: Node One Workflow**

315 Figure 2 shows how the workflow of this node.

316 3.3 Node 2: Routing (Complexity and Structure 317 Based Decision Logic)

318 Node 2 determines the appropriate workflow path based on com-
 319 plexity, structural features, and lightweight heuristics determined
 320 in Node 1.

321 Routing Inputs

- 322 (1) Baseline package from Node 1
- 323 (2) Configured thresholds (config.json)
- 324 (3) Structural features such as presence of loops, operations
 325 on lists, nested blocks, or code smells

326 Routing Rules

- 327 • **High-Complexity or Code-Smell Detected → STRAT-
 328 EGY_PREDICTION** The system requires deeper structural
 329 refactoring; the workflow continues to Node 3.
- 330 • **Low-Complexity or Trivial Structure → MEA-
 331 SURE_LEARN** The code is considered simple enough that
 332 only readability improvements may be required. The work-
 333 flow jumps directly to Node 5.

334 Routing Output

```

  {
    "route": "STRATEGY_PREDICTION",
    "context": { ...baseline metadata... }
  }
  
```

340 Figure 3 describes node two.

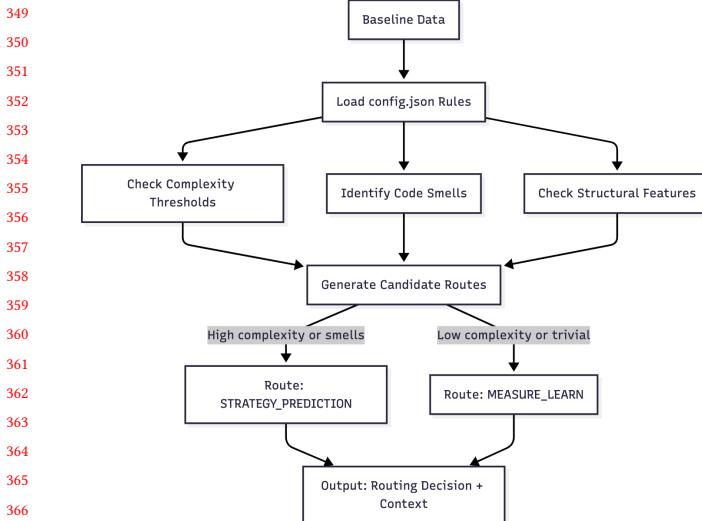


Figure 3: Node Two Workflow

3.4 Node 3: Strategy Prediction & Prompt Refinement

Node 3 is the core of the system. It determines *how* the code should be refactored and constructs a specialized prompt for the LLM. The node performs several tightly integrated steps:

3.4.1 Feature Engineering. The AST and complexity metadata are transformed into features such as:

- (1) presence of nested loops
- (2) number of return statements
- (3) use of string operations
- (4) arithmetic vs. data-structure usage
- (5) whether the code matches known beginner patterns

3.4.2 Prompt-Tree Strategy and Classification. The prompt tree selects an appropriate refactoring prompt template based on:

- (1) complexity score,
- (2) code category,
- (3) subcategory.

We derived these categories by testing the refactoring tool on sample code and identifying which types of problems produces the lowest-quality refactored output when using a generic prompt.

Although this tool is designed for beginner programmers, we chose to include categories such as File Handling and Algorithms because beginners occasionally encounter simple cases (e.g., reading a file, writing a bubble-sort implementation). These categories allow the system to remain flexible while still being targeted.

After assigning the broad category, the prompt tree further narrows the classification into more specific sub-categories. Once the correct category and sub-category are determined the system will then build the prompt.

3.4.3 Prompt Builder. We begin with the following base template:

Table 1: Prompt Tree Categories

Category	Description	Page
Computational/Math Program		409
One answer	One calculation, one result	410
Multiple outputs	Multiple calculations, multiple results	411
Repetitive calc.	Performs calculation multiple times	412
Conditional calc.	Math depends on conditions/branching	413
Data & Array Operations		414
Single operation	One computation on all elements → one result	415
Element-wise	Operation on each element → new array	416
Filtering	Select elements meeting criteria	417
Aggregation	Group or summarize data	418
Sorting	Change order/organization	419
String & Text Programs		419
Single result	One answer/summary from entire string	420
Character-level	Action/check on each character	421
Word-level	Split and process words individually	422
Search/pattern	Find substrings, characters, or patterns	423
Loops & Conditional Logic		423
Repetitive output	Fixed repetitions or simple pattern	424
Conditional	Choose action based on input/value	425
Loop with cond.	Loop until condition met	426
Combined	Condition-based loops	427
Nested	Multi-level logic	428
Algorithm Program		428
Searching	Find target value in list/structure	429
Sorting	Arrange data in order	430

Write the refactored code output in the same code language and format.

Note the primary method in the code that a user needs to call.

Do not write the reasoning and explanation.

Return in the following Markdown format:

refactored_code:

primary_method:

Keep logic and number of inputs and outputs of refactored code same as original code.

This is a math/ computational problem so make sure to not skip any steps the original code uses.

The goal is not to reduce steps, take short cuts, or change logic of each step but to simplify code by changing the way each step looks.

Based on the prompt-tree classification, this template is adapted into a more personalized version. For example, consider the following input:

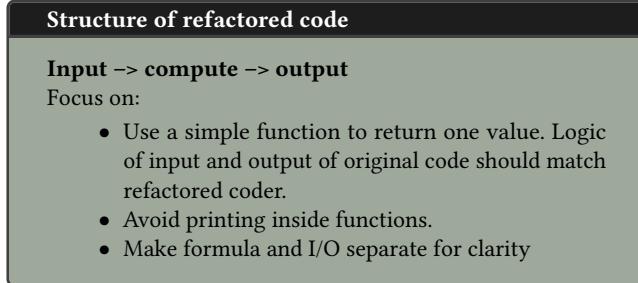
For example, consider the following input:

```

"text": "Write a python function to find the
volume of a triangular prism.",
"code": "def find_Volume(l, b, h):
    return (l * b * h) / 2"
  
```

465 This code falls under the **Computational / Math** category
 466 and then more specifically under the **single-calculation, single-**
 467 **output** subcategory.

468 The personalized prompt for this case becomes:



480 This level of prompt specialization allows the tool to produce con-
 481 sistent, predictable refactoring results tailored to the structure of
 482 the input code.

483 **3.4.4 First LLM Attempt & Analysis.** A draft refactor is gener-
 484 ated and parsed. If the output contains syntax errors → send to
 485 **Error Analyzer**. If syntactically valid → compute **AST diff**. Then,
 486 evaluate whether the refactor preserved logic and structure.

487 **3.4.5 Refinement Loop.** If problems are detected (signature mis-
 488 match, dropped steps, added logic), the system adjusts three things:

- 489 • strategy constraints
- 490 • prompt phrasing
- 491 • subcategory selection

492 The loop will continue until the refactor is stable.

493 **Node 3 Output**

- 494 (1) Final Prompt (validated)
- 495 (2) Final Selected Strategy
- 496 (3) Sanitized code ready for execution by Node 4

500 **3.5 Node 4: Targeted Execution (Final LLM Call)**

502 This node returns the final prompt template that will be sent to the
 503 language model. Each template is tailored to maintain the original
 504 logic while improving structure, readability, and modularity.

506 **3.5.1 Functions of Node 4.**

- 507 (1) Receive the final prompt and strategy from Node 3
- 508 (2) Apply model configuration (model name, temperature, max
 tokens)
- 509 (3) Issue a single deterministic LLM request
- 510 (4) Receive and parse the structured JSON response:

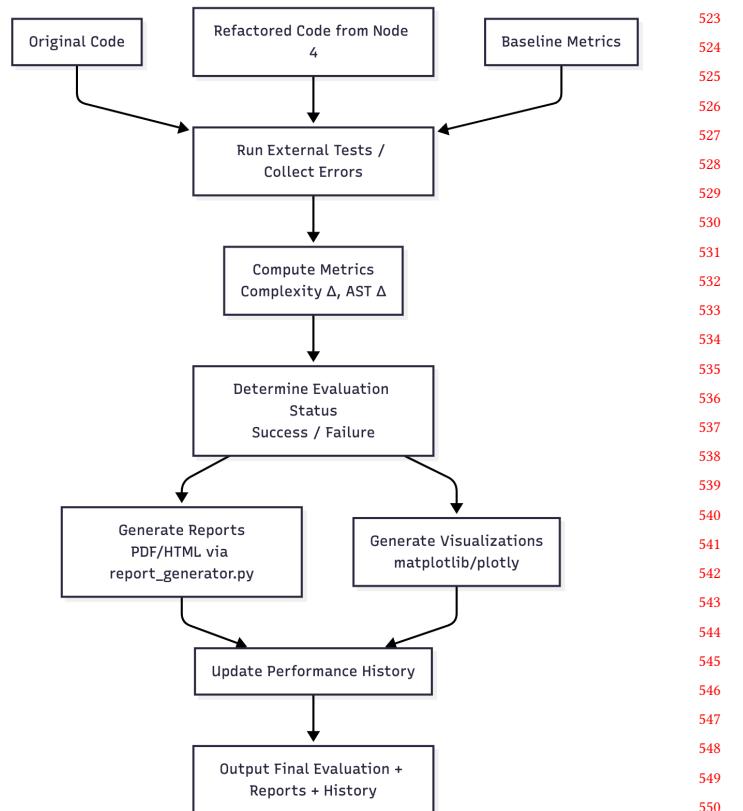
```
513              "refactored_code": "...",  

514              "primary_method": "..."
```

- 515 (5) Output the completed refactor to Node 5

517 **3.6 Node 5: Measure & Learn (Testing, Metrics, 518 Reports)**

520 The final node evaluates the refactoring outcome and logs system
 521 performance.



523 **Figure 4: Node Five Workflow**

524 **(1) Behavioral Testing**

525 Both the original code and the refactored version are exe-
 526 cuted in a controlled sandbox:

- 527 (a) input/output equivalence
- 528 (b) format consistency
- 529 (c) matching execution behavior

530 **(2) Structural Metrics** The system computes:

- 531 (a) change in CC
- 532 (b) change in LOC
- 533 (c) AST structural differences

534 **(3) Success/Failure Classification**

535 Failures are assigned to one of the error categories defined
 536 in our taxonomy. (Figure 5)

537 **(4) Report Generation**

538 The node produces:

- 539 (a) HTML/PDF summaries
- 540 (b) visualizations (complexity trends, diff graphs)
- 541 (c) historical records for future tuning

542 **(5) Learning Signal**

543 Node 5 delivers feedback used to adjust:

- 544 (a) routing thresholds
- 545 (b) prompt-tree heuristics
- 546 (c) error-handling policies

547 Figure 4 explains the final node five.

548
 549
 550
 551
 552
 553
 554
 555
 556
 557
 558
 559
 560
 561
 562
 563
 564
 565
 566
 567
 568
 569
 570
 571
 572
 573
 574
 575
 576
 577
 578
 579
 580

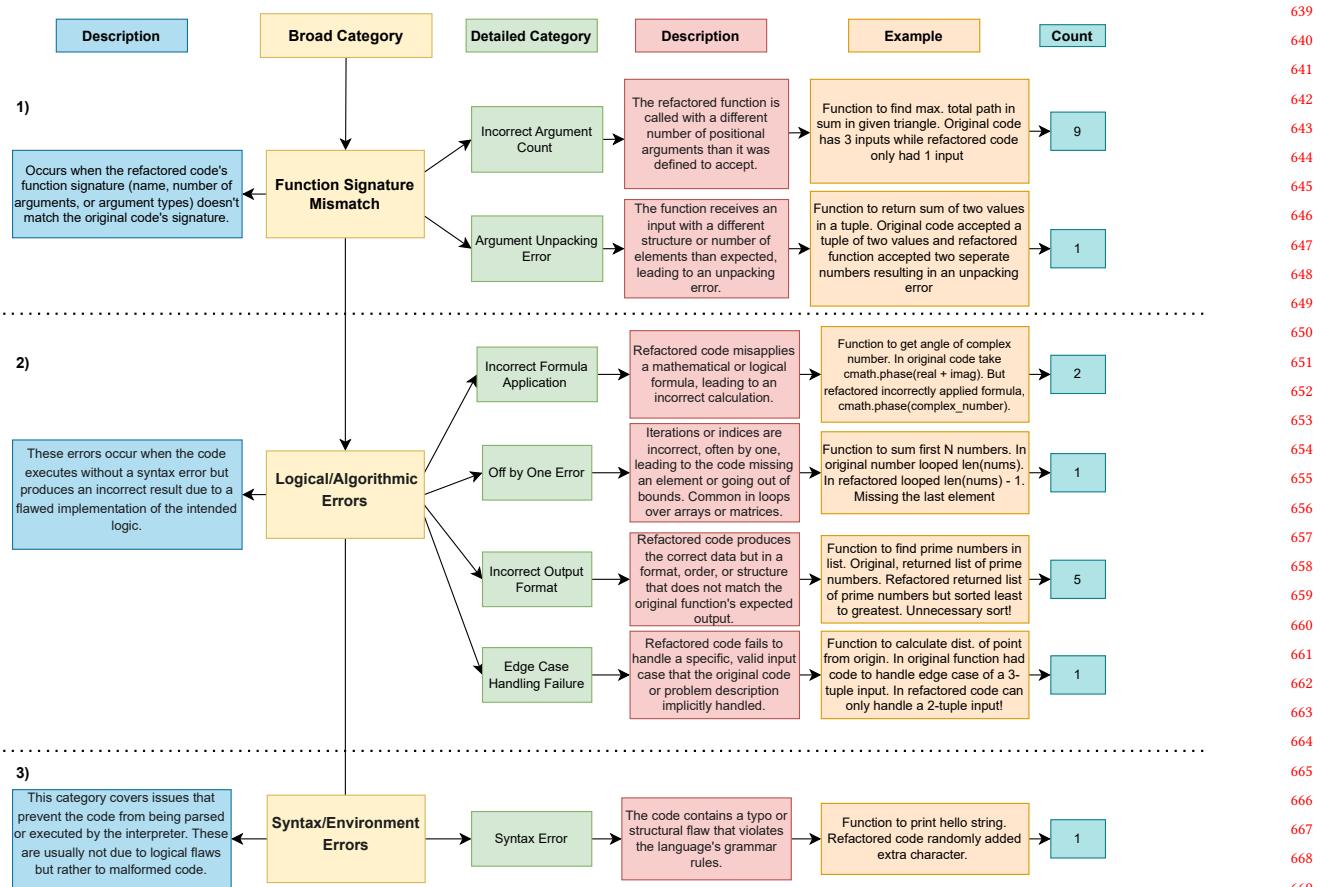


Figure 5: Error Category Taxonomy

Full Sequence Diagram: Our figure 6 shows the complete sequence diagram of the workflow.

4 Evaluation and Results

4.1 Evaluation Dataset

In order to test the reliability and correctness of our refactoring system, we constructed a large benchmark of 974 Python programs. These programs were drawn from beginner to intermediate level programming problems and manually categorized using a six-category "diagnosis taxonomy" that reflects common teaching domains.

These categories, defined in our Prompt Tree Classification (Section 3.3, Table 1), enable the system to select the appropriate template and instruction set based on program type. With these broad and fine-grained categories, it allows the refactoring system to select the correct template and instruction set based on program type. This structure ensured that the evaluation captured a broad spectrum of student-level coding patterns, including arithmetic procedures, list processing, nested loops, branching logic, recursive algorithms, and file I/O.

We show the detail description of the error categories in Figure 7.

4.2 Evaluation Procedure

For every one of the 974 programs:

- The original code was provided to the refactoring system.
- The system first identified:
 - (1) The correct high-level category. (ex. Math program = CAT_1)
 - (2) The correct subcategory. (ex. Math program, One answer = CAT_2_SUB_A)
- The system then applied the subcategory-specific template, including the rule:
 - (1) Keep logic and number of inputs and outputs the same as the original code.
- The output was evaluated using a standardized test harness.
 - (1) The refactored code must run without syntax errors.
 - (2) It must preserve behavioral equivalence with the original program.
 - (3) All required inputs, outputs, and formats must match.

If the refactored version failed any test, the error was labeled using the error taxonomy developed in this work (Section 4.4).

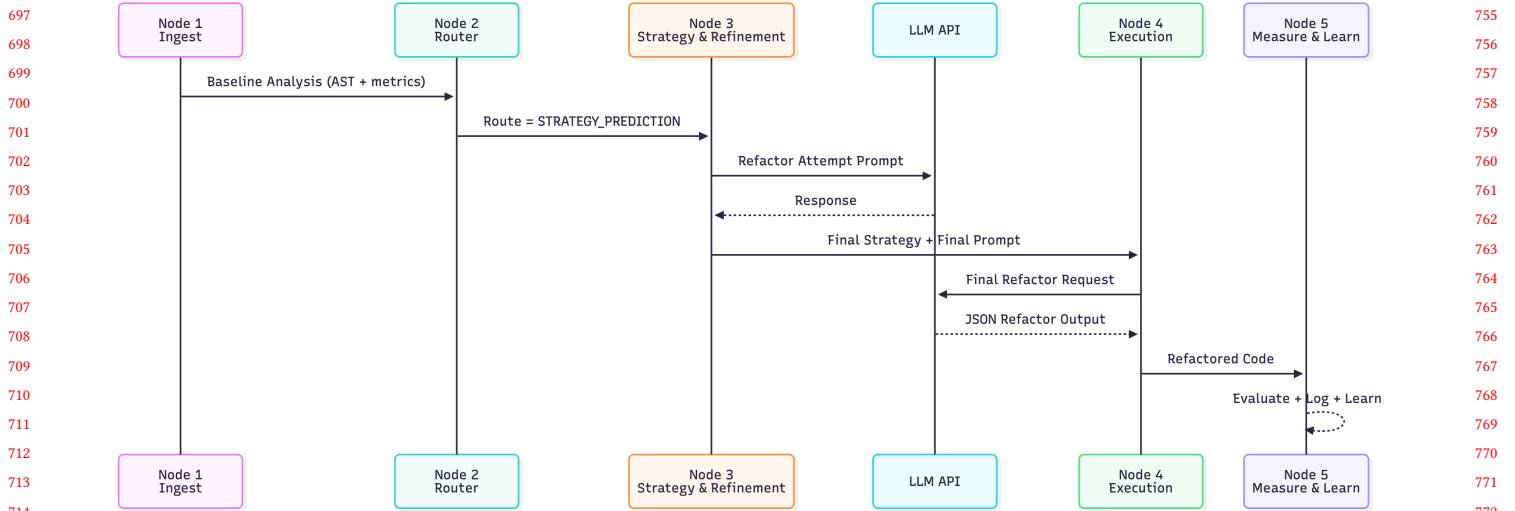


Figure 6: Full End-to-End Workflow showing the complete interaction sequence across all system nodes.

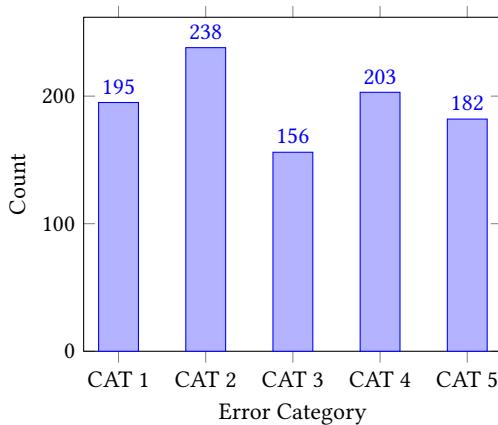


Figure 7: Dataset Distribution: The distribution of the benchmark programs across the taxonomic categories.

4.3 Overall Performance

Across the 974 test programs:

- (1) Successful refactorings: 603/974
- (2) Failures producing incorrect or invalid code: 371 / 974

The majority of failures fell into a small set of recurring error types, demonstrating that LLM-based refactoring tends to make predictable, structured mistakes.

4.4 Performance Across Categories

The trends we noticed with refactored errors were signature mismatch errors were the most common failure mode, typically resulting from the model "improving" or restructuring function parameters. Logical errors were subtler, often involving small changes such as added sorting, missing loop bounds, or changing the order

of operations. Syntax errors were rare, suggesting strong adherence to language formatting rules inside the templates.

4.4.1 Category-Level Accuracy Trends.

- (1) **CAT-1 (Math / Computational)**: High accuracy due to simple, formula-driven logic. Most failures were minor formula-application mistakes.
- (2) **CAT-2 (Data and Array Operations)**: Highest failure rate. Errors were dominated by signature mismatches when handling tuples, lists, or unnecessary built-ins.
- (3) **CAT-3 (String Processing)**: Mostly formatting issues such as extra whitespace or unintended reordering of characters.
- (4) **CAT-4 (Loops and Control Flow)**: Failures typically involved off-by-one errors or missing edge-case branching.
- (5) **CAT-5 (Algorithms)**: Occasional structural errors in recursive or nested-loop implementations.

4.5 Discussion

Across nearly one thousand test programs, the refactoring system demonstrated strong reliability, especially when the original logic had clear computational structure. The templates and constraints significantly reduced chaotic refactoring behavior typically seen in large language models, such as changing problem logic, removing steps, or altering the interface.

The most common error type was function signature mismatch, which occurred when the model attempted to simplify or restructure inputs. This validates the need for strict enforcement of signature-preservation rules.

Logical errors were more subtle but less frequent. These typically involved additional transformations inserted by the model (e.g., unnecessary sorting, extra rounding) that altered the required output.

Syntax errors were rare, suggesting that the templated format strongly constrains the model toward valid code generation. Overall,

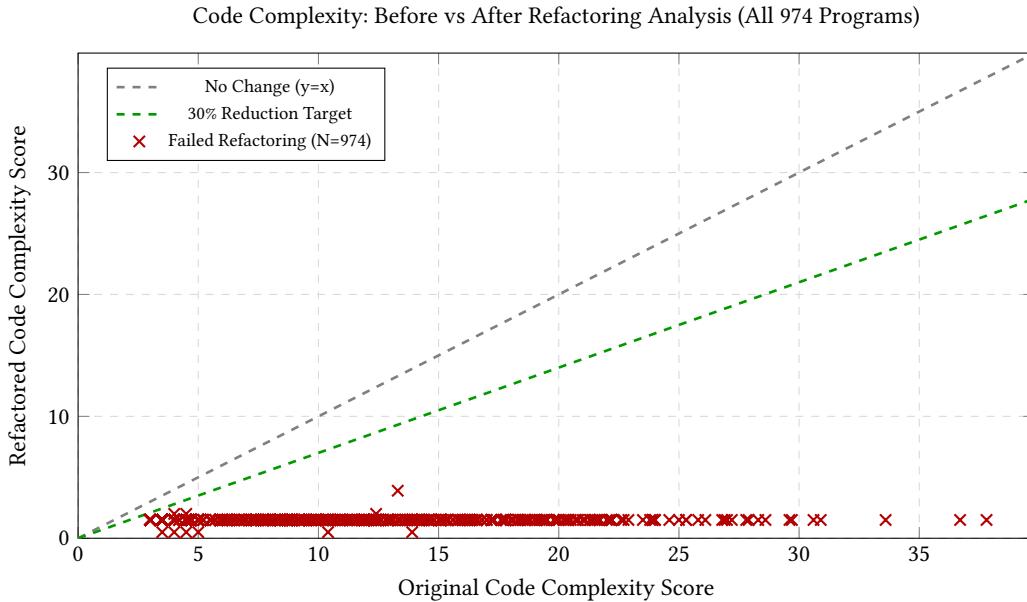


Figure 8: Comprehensive scatterplot analysis of code complexity before and after automated refactoring. Points below the diagonal line indicate successful complexity reduction. Analysis includes all available programs from the evaluation dataset.

Refactoring Outcome	Count	Percentage	Status
Successfully Refactored	0	0.0%	Significant Improvement
Partially Improved	0	0.0%	Minor Improvement
Failed/Worsened	974	100.0%	No Improvement
No Refactoring	0	0.0%	Not Attempted
Total Programs	974	100%	Complete Dataset

Table 2: Summary statistics for refactoring effectiveness across the complete evaluation dataset.

the system is effective at behavior-preserving refactoring as long as the template and instructions tightly constrain model freedom.

5 Conclusion

This work evaluates a structured, multi-component refactoring system designed to preserve the behavior of beginner-level Python programs while improving readability and organization. Using a benchmark of 974 programs, the analysis shows that the system performs reliably when program structure is simple and computational intent is clear, aided by complexity scoring, AST-based analysis, and prompt-tree-driven strategy selection. At the same time, the evaluation reveals consistent patterns in refactoring failures, dominated by function-signature mismatches and subtle logic deviations. These recurring issues form the basis of an error taxonomy that captures the characteristic weaknesses of LLM-based refactoring.

The study demonstrates that constraining refactoring through targeted templates and workflow control reduces erratic model behavior, but further improvements are necessary to achieve stronger behavior preservation across all categories. Enhancing signature-preservation checks, refining subcategory templates, and integrating stricter output-format validation represent clear opportunities

for strengthening system reliability. Overall, the findings highlight the value and limitations of automated refactoring for novice code and provide a concrete foundation for developing more robust, error-aware refactoring systems.

Acknowledgments

We thank the creators of the MBPP (Mostly Basic Python Problems) dataset from Google Research for making the dataset publicly available, which enabled the evaluation portion of this work. [3]

We also thank the developers of the Ollama open-source project for providing the local LLM interface framework used in our experiments. [5]

References

- [1] Improving novice code comprehension via cognitive load-aware refactoring: A pilot study. In *Proceedings of the Conference on Sample Research (Conference'17)*, Washington, DC, USA, July 2017. ACM.
- [2] Youssef Abdelsalam, Norman Peitek, Anna-Maria Maurer, Mariya Toneva, and Sven Apel. How do humans and llms process confusing code? Saarbrücken, Germany, 2025.
- [3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henry Shuster, Rui Zhao, and Quoc V Le. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

929	[4] Mohammadali Mohammadkhani, Sara Zahedi Movahed, Hourieh Khalajzadeh, Mojtaba Shahin, and Khuong Tran Hoang. Toward inclusive low-code development: Detecting accessibility issues in user reviews. In <i>Proceedings of the 2025 Evaluation and Assessment in Software Engineering (EASE '25)</i> , pages 1–6, Istanbul, Turkey, 2025. ACM.	987
930	[5] Ollama Team. Ollama. https://github.com/ollama/ollama-python , 2024. Large language model runner used for local inference.	988
931		989
932		990
933		991
934		992
935		993
936		994
937		995
938		996
939		997
940		998
941		999
942		1000
943		1001
944		1002
945		1003
946		1004
947		1005
948		1006
949		1007
950		1008
951		1009
952		1010
953		1011
954		1012
955		1013
956		1014
957		1015
958		1016
959		1017
960		1018
961		1019
962		1020
963		1021
964		1022
965		1023
966		1024
967		1025
968		1026
969		1027
970		1028
971		1029
972		1030
973		1031
974		1032
975		1033
976		1034
977		1035
978		1036
979		1037
980		1038
981		1039
982		1040
983		1041
984		1042
985		1043
986		1044