Refactoring Codebases through Library Design

Žiga Kovačič[*]
Cornell University
zk66@cornell.edu

Celine Lee[*]
Cornell University
cl923@cornell.edu

Justin T Chiu[*]
Cohere
chiu.justin.t@gmail.com

Wenting Zhao[†]
Cornell University
wz346@cornell.edu

Kevin Ellis[†]
Cornell University
kellis@cornell.edu

Abstract

Maintainable and general software allows developers to build robust applications efficiently, yet achieving these qualities often requires refactoring specialized solutions into reusable components. This challenge becomes particularly relevant as code agents become increasingly accurate at solving isolated programming problems. We investigate code agents' capacity to refactor code in ways supporting growth and reusability. We present both a method and a benchmark for refactoring: LIBRARIAN, a sample-and-rerank method for generating reusable libraries, and MINICODE, a benchmark where code agents must minimize and refactor multiple independent solutions into a joint library. Compared to state-of-the-art code agents, LIBRARIAN achieves strong results on both compression and correctness on MINICODE, obtaining compression rates 1.6-2x better than coding agents while also improving correctness. We open-source our code and benchmark at `https://code-refactor.github.io/`.

1 Introduction

Writing code is mainly a matter of *re*writing code: debugging, refactoring, optimizing, and other activities within the software engineering lifecycle. But poor rewrites incur technical debt, with such debt costing up to $2 *trillion* annually [1]. This problem will likely worsen as language models become increasingly responsible for generating code, because they excel at solving isolated programming problems, but their context length demands a myopic view of the codebase. It is therefore valuable to understand not just the ability of language models to solve programming problems, but also their ability to rewrite and refactor code in ways that support growth and reuse.

Effective code refactoring at scale is a design problem. When refactoring codebases, developers must navigate design decisions around concerns such as generality, re-usability, and maintainability. A classic example illustrates this design challenge: Human programmers often create overly-specialized, redundant solutions to similar problems and would benefit from redesigning specialized solutions into a shared library. This consolidation requires careful design decisions about the right level of abstraction — neither too specific nor too general — and appropriate interfaces that balance flexibility with usability.

Here we focus on refactoring multiple code sources into a reusable software library, and pose the following question: To what extent can code agents address this problem, both within human-written codebases, and also in language model-generated code? To answer that question, we develop a new

[*]denotes equal contribution.
[†]denotes equal advising.

Preprint. Under review.

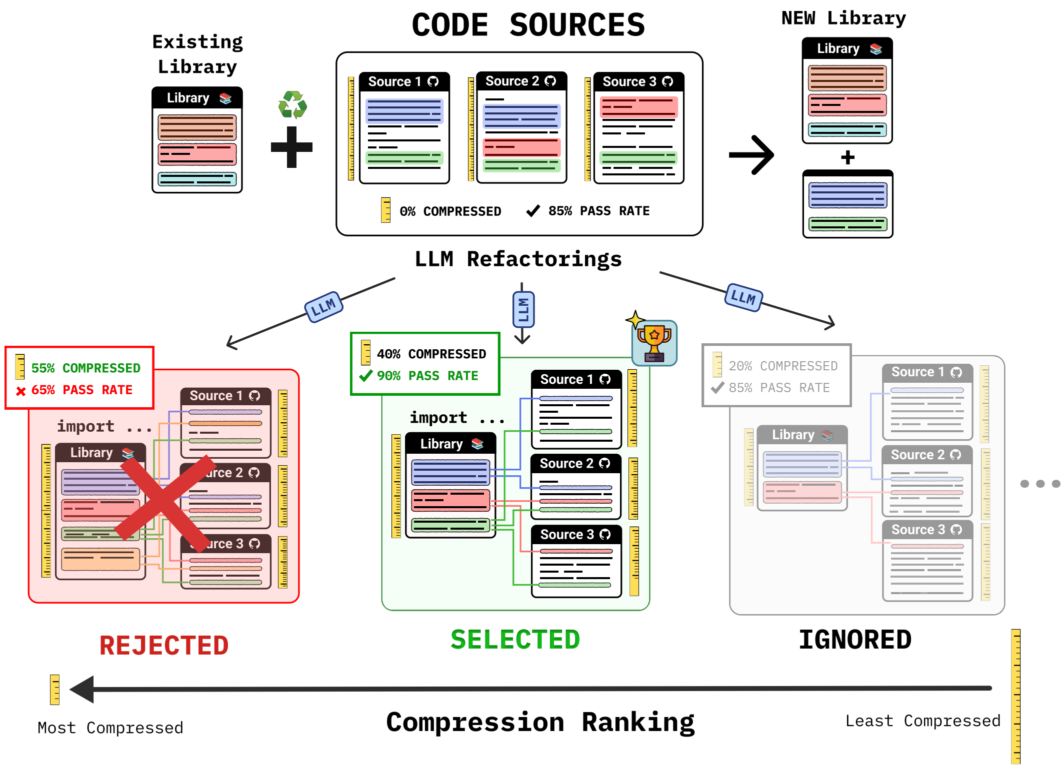

Figure 1: Overview of the problem that we study and the general structure of its solutions. Given a collection of different code source, where a source is either program or repository,—and optionally an existing library—we refactor the code sources by designing a new modular and reusable library. Candidate refactorings are evaluated based on program simplicity (compression), and are expected to maintain correctness of the original code sources (pass rate).

method and a benchmark. This goes beyond past work [2, 3, 4, 5, 6, 7, 8] in *library learning* that synthesized subroutines across small programs in i.e. λ-calculus, instead tackling the more naturalistic problem of redesigning large bodies of code written in contemporary high-level languages, such as Python, producing classes, methods, and helper functions in the style of a human-written library. We propose a simple method, LIBRARIAN (Figure 1), which samples possible code rewrites then reranks those samples based on criteria designed to capture what it means to have a good refactoring. To generate potential rewrites, we develop methods for clustering pieces of code together that share common structure, so that a succinct prompt can rewrite them jointly into their refactored form.

To evaluate our method and systematically assess the capability of current agents to perform such design-intensive refactorings, we introduce a new benchmark, MINICODE, which addresses three key desiderata missing from existing benchmarks. First, open-ended design: unlike SWE-Bench [9], Commit0 [10], and RefactorBench [11] which primarily focus on functional correctness, MINICODE presents an unconstrained library design problem. Agents must create a library that can be imported into multiple repositories, with complete freedom to design the interface and implementation from scratch—optimizing for software engineering objectives like reusability and maintainability. Second, verifiability: we ensure objective evaluation by retaining the unit tests from all repositories that will import the designed library, allowing us to verify that the solutions work correctly across multiple use cases. Third, large context: agents must understand and synthesize information from multiple repositories simultaneously to design a unified library that consolidates specialized code sources into a general interface. Prior benchmarks typically focus on single-repository tasks.

Our results show that state-of-the-art code agents, based on Claude 3.7 Sonnet and o4-mini, struggle to jointly preserve correctness and improve reusability across both domains of MINICODE. In the competition coding domain, our method LIBRARIAN improves refactoring quality by 1.89x while also enhancing correctness. However, on the repository-level refactoring, even the strongest agents fail to produce high-quality refactorings, highlighting a substantial gap between current capabilities and the demands of design-oriented code rewriting. Addressing this challenge remains an open and important direction for future research.

2 Related work

Repo-level coding benchmarks. Recent work has explored the application of language models to repository-level software engineering tasks. Existing benchmarks include SWE-bench [9], which evaluates models on their ability to resolve real-life GitHub issues, and Commit-0 [10], which requires agents to fill in function definitions. Such benchmarks primarily evaluate functional correctness via unit tests, without assessing the quality or maintainability of the resulting codebase. Refactor-Bench [11] takes a step in this direction by benchmarking the ability to follow specific refactoring instructions. Our work differs by requiring models to perform a more open-ended task: Redesigning code to be more modular and compact by discovering and drawing out reused abstractions, while retaining verifiability by re-using downstream unit tests. Additionally, libraries must be created without any scaffolding limitations such as preexisting function definitions affording more design freedom than Commit-0.

Library Learning. Systems which perform library learning research discover shared abstractions across a large number of small programs, which they use to automatically define new subroutines. Systems such as DreamCoder [4], Trove [12], LiLo [7], and REGAL [3] automatically construct such libraries with the goal of making future program synthesis tasks easier to solve, once the learned library is in hand. Our work is closest to REGAL [3], which clusters related code and refactors using language models. However, existing library learning approaches have primarily been demonstrated in small-scale, constrained domains, limiting their applicability to typical software engineering tasks, such as consolidating multiple repositories into cohesive libraries. By framing library learning within the context of realistic, large-scale code repository development, we expand the relevance of library learning to everyday software engineering practice.

Program optimization. While our goal is to optimize the quality of libraries, other works focus on improving execution speed through correctness-preserving transformations [13, 14, 15]. Both forms of program optimization, compression and speed, are more open-ended than optimizing only for correctness, as there does not exist a ground-truth answer. Prior work on program optimization benchmarks study code at the file level. We propose a benchmark that transforms programs at a larger scale, across multiple code repositories.

3 Problem Statement

In this section, we propose a refactoring task: Given multiple code sources that contain problem-specific implementations, the goal is to create a cohesive library that captures shared abstractions. This library must reduce the total code size while supporting all original use cases, potentially opening up new use cases as well by mining and formalizing latent shared abstractions. This is accomplished by searching for refactorings that are both correct and simple. Correctness is straightforward to define as the fraction of unit tests passed, but simplicity is more elusive.

One potential measure of simplicity is counting the total number of tokens in the proposed library and refactored code. However, just minimizing program size has obvious failure modes: code should also be natural, elegant, and extensible, which can be in tension with merely finding the shortest program.[3] To address these concerns, we follow prior work in program synthesis [16, 6, 17, 18] and define simplicity as the *minimum description length* (MDL), or negative log probability under a reference distribution.

Formally, we are given a collection of code sources $\{\rho_n\}_{n=1}^N$, and output both a new library $\mathcal{L}$, as well as rewritten refactorings of the original code sources, $\{\rho'_n\}_{n=1}^N$. We define the pass rate $\tau(\rho_n)$ as the fraction of unit tests program ρ_n passes. In practice we are concerned both with the case where

[3] Perl Golf is a game where participants attempt to write the shortest Perl program accomplishing a given task. The resulting code is famously incomprehensible, even by the standards of Perl.

we are refactoring several code sources ($N > 1$) and also the case where there is only a single large code source we are refactoring ($N = 1$).

We optimize the following objective, which rewards refactorings that pass at least as many tests as the original program *and* minimize MDL:

$$\ell(\mathcal{L}, \{\rho'_n\}) = \begin{cases} -\log p_{\mathrm{LM}}(\mathcal{L}) + \sum_n -\log p_{\mathrm{LM}}(\rho'_n \mid \mathcal{L}) & \forall \rho_n, \tau(\rho_n) \le \tau(\rho'_n) \\ \infty & \text{otherwise} \end{cases} \tag{1}$$

where $p_{\mathrm{LM}}(\rho'_n|\mathcal{L})$ is the probability of the suffix ρ'_n given the prefix $\mathcal{L}$ under a language model, effectively concatenating the library and the program into one prompt, but only counting the perplexity of the later program tokens.

4 LIBRARIAN: Refactoring Code to Create Libraries

This section details our method to compress collections of code sources into libraries, while migrating the code sources to use these shared building blocks. Figure 1 illustrates our method, LIBRARIAN.

LIBRARIAN follows a simple sample-and-rerank framework to maximize our refactoring objective described in Section 3. It maintains and grows a library of useful functions as part of this objective.

Concretely, our framework follows:

$$\mathcal{L}^\star, \{\rho^\star_n\} = \operatorname*{arg\,min}_{\mathcal{L}, \{\rho'_n\} \in \textsc{Sample}(\{\rho_n\})} \ell(\mathcal{L}, \{\rho'_n\}) \,. \tag{2}$$

4.1 Sample with clustering

Meaningful abstractions exist primarily among programs that share some functionality or underlying structure. We perform clustering on the input programs to make groups of programs that likely share structures that can be abstracted into general library functions. Most modern language models cannot be prompted with the entire collection of input programs– even long context models cannot process the entirety of e.g the Linux kernel, and even if they could, it is not clear that such a strategy is the most efficient way of focusing the language model's attention.

We consider clustering algorithms for discovering small groups of related code; we call these *tuples*. This extends REGAL [3], which clusters programs solving similar problems by assuming each program is paired with a natural language description of the problem it solves, and clustering embeddings of those descriptions. But programs solving similar problems do not necessarily have similar structure. We therefore instead first summarize the code source itself by prompting a language model, then cluster based on the similarity of these new descriptions.

Once identified, each tuple is used in two stages. The first is to retrieve relevant already-abstracted functions from the LIBRARIAN library: for a given tuple of programs, relevant functions are retrieved by prompting a language model with the entire existing existing library and the original input programs, then instructing it to identify which functions would be useful. The retrieved library functions and the original programs in the tuple are then provided as critical context to the language model to propose a sample budget of K candidate refactorings.

4.2 Rank with compression

Once sampled, all K candidate refactorings are passed through a *sort-and-filter* evaluation harness to select which one scores the highest on refactor quality and that maintains (or improves) test accuracy compared to the original. If no such candidate exists, the original code is preserved, maintaining existing functionality.

New library functions in the selected refactor are saved into the LIBRARIAN library for potential use in downstream refactoring of other programs. We provide the full algorithm in Appendix A.

5 MINICODE

MINICODE evaluates a code agent's capability to identify abstractions across implementations and design reusable libraries. In order to measure these capabilities, our benchmark presents agents with a

Table 1: MINICODE Statisics

Domain	Sources	Collections	Avg LoC	Avg Tests	Gen by
Code Contests	300	30	87	10	Humans
Small Repositories	262	22	209	12	o4-mini
Large Repositories	20	10	6,433	101	Claude-Sonnet 3.7

collection of code sources, then asks agents to refactor the code sources into a unified library alongside refactorings of the original code sources. There are two key desiderata for collections of code sources: The collections must be compressible, in that there exists a latent shared library abstraction, and verifiable, so that we can measure how well refactored code sources preserve functional correctness. We source problems from two domains: Competition coding and synthesized repositories (Table 1).

Agents are expected to interact with MINICODE via the terminal. We structure the benchmark as refactoring a multi-package Python repository, where each code source in a collection is a Python package in a subdirectory. This requires knowledge of basic bash commands for exploring repositories, editing code, and running tests, as well as how to manage complex, multi-package Python libraries.

CodeContests Competition problems are crafted with specific variations of algorithmic approaches in mind, resulting in both shared latent concepts and required test cases. As a result, competition coding is naturally both compressible and verifiable.

Each collection consists of multiple code sources, each containing a solution to a competition programming prompt, with associated tests for verification. We take solutions, prompts, and tests from CODECONTESTS [19], a dataset consisting of competition coding problems. Each code source in the collection is structured as a subdirectory consisting of the task description in `PROBLEM.md`, the initial solution in `main.py`, and a script to run tests in `run.sh`. Agents are instructed to create a `library.py` file, which should is imported into each code source. Since CODECONTESTS has no external dependencies on Python packages, this can be done without explicit structuring as a Python package.

Repositories For the second domain of synthesized repositories, we propose a data-generating process that first produces project ideas, then generates variations of those project ideas tailored to specialized use-cases. This allows us to control the complexity and degree of overlap between each code source in a collection. Each code source in the repositories domain is comprised of a task description, source code, and test cases for functionality and correctness. MINICODE includes both small repositories (approximately 200 lines of source code each) and large repositories (approximately 6.5k lines of source code each), both of which represent realistic settings where different people with different needs use language models to help them write software for their particular use cases. The refactoring agent is tasked with extracting re-usable functions from across code repositories, and re-writing the original code source repositories to use them.

We approximate the true distribution of code repositories, $p(\rho)$, with a generative process using latent textual library and repository descriptions. The entire space of code repositories is massive. To collect and group code repositories that share meaningful structure that can be abstracted into a useful code library, we (1) sample a textual library description; (2) sample use cases; and last (3) their programmatic implementations. This generative process naturally produces repository collections primed for refactoring. We sample from language models for each step of this process. Prompts are shared in Appendix E.

Each collection consists of multiple synthetic repositories as code sources, where each code source is a subpackage. Collections are obtained by transforming the original repository code sources into a multi-package Python library. Each source's source code is extracted into a subpackage directory, and its tests into a corresponding test subdirectory. Agents are instructed to write a shared library in a shared subpackage named `common`. This `common` shared library must be imported and used to refactor each of the original code sources.

6 Experimental Setup

Grouping Programs into Collections To facilitate parallel application of LIBRARIAN and manage the dataset scale, we assume that semantically distant code sources will have minimal overlap in their optimal library functions. Therefore, our overall approach partitions the dataset into disjoint collections through clustering.

For **CodeContests**, these collections are constructed from an initial corpus of ∼9k problems with Python solutions: We first filter these code sources, removing those whose selected canonical solution is under 10 lines (minimal refactoring potential). For the remaining 4596 solutions we use a language model to generate textual descriptions of canonical solutions—emphasizing reusable components—which are embedded using OpenAI's `text-embedding-ada-002`.

Agglomerative Clustering [20] is subsequently applied to these embeddings to partition the code sources into a predefined number of initial clusters, in our case 120. To create uniformly sized experimental units, we subsample each such cluster to form collections of 30 code sources. This collection size was empirically chosen because it balanced between the runtime of LIBRARIAN without limiting compression. We select 10 collections that we then use to evaluate our methods.

Code repositories are generated as disjoint collections through the generative process, which first samples project ideas then variations. For small repositories, we up sample $10 - 15$ code sources then form smaller collections of size 3-5 code sources that exhibit the most intra-cluster similarity according to clustering using the same embedding and clustering technique. For big repositories, we sample two code sources for each of the 10 collections. Small repositories do not rely on Python packaging. Large repositories are setup as multi-package monorepositories. These monorepos are constructed by merging the dependencies and tests of the initial packages, and placing the source directories into the root directory of the monorepo.

REGAL Baselines To evaluate the ability of our libraries to support reuse on new problems, we turn to the program synthesis tasks used in REGAL, where learned libraries are added to help the program synthesizer. We evaluate on the two domains published by the authors, Logo and Date. Because our clustering is inspired by REGAL but adds additional complexity, for fair comparison, we keep their setup the same and only augment the training using sample + MDL rerank procedure described in Section 4.1.

Code Contests To evaluate LIBRARIAN on refactoring Code Contests we select 6 collections of 30 code sources (problems). In each collection we group the problems into tuples of size 3. We set the sample budget to be $K = 8$, since our ablations show that with larger K we discover better libraries 3. We use the MDL objective for rankings.

The model used for sampling is OpenAI's o4-mini [21]. To obtain MDL scores we use Qwen 2.5 7B Instruct [22] as a balance between quality, speed, and cost.

Code Agents To fairly evaluate performance on the task by state-of-the-art systems, we use coding agents that advertise long-context ability to reason about, write, and refactor code repositories. Specifically, we use Claude Code (Cl) [23] which uses the Claude 3.7 Sonnet model, and OpenAI's Codex (Cx) [24] which uses o4-mini.

All steps of the initial code sources for small code repositories are done by OpenAI Codex. We empirically observed that Claude Code has an affinity for generating large code repositories– the large code repositories are all generated by Claude Code.

We test whether code agents can refactor collections of code sources autonomously, without human intervention. Refactoring repositories with code agents involves planning and iterative (re-)implementation and testing. Code agents are prompted to perform each of these steps, with feedback from the unit tests. Agents must run and repair unit tests autonomously. We run coding agents multiple times per collection, logging their progress in checklists stored in text files. Our naming convention for agents is "Planner-Executor". For example, "Cl-Cx" uses the Claude Code agent for planning and OpenAI Codex to implement the plan.

As before we use Qwen 2.5 7B Instruct [22] to obtain MDL scores.

7 Results

Metric	Value
Pass Rate	90.67% ±1.88
Pass Rate Improvement	6.33% ±1.41
MDL Ratio	0.53 ±0.03
Token Ratio	0.66 ±0.04
Library Functions	10.30 ±1.41
Avg Calls per Function	5.17 ±1.08
% Single Use Functions	38.03% ±4.88

Figure 2: Refactoring results for LIBRARIAN (w/ $K = 8$) averaged over 10 Code Contests collections.

In this section, we present the compression and correctness results with LIBRARIAN and agent baselines on MINICODE.

MINICODE-CodeContests On CodeContests, **LIBRARIAN** achieves a high final pass rate of 90.67% and significantly improves correctness, with pass rates increasing by 6.33% compared to the original code sources (Table 2). The method yields substantial compression: the refactored code, including the new library, shows an MDL ratio of 0.53 (a 47% reduction in MDL relative to the original). On average, LIBRARIAN generates libraries containing approximately 11 functions. These functions demonstrate good reuse, being called by around 5.2 programs on average, although 38.03% of them are used only once within their specific collection context.

Code agents fail to achieve both high correctness and compression on MINICODE-CodeContests. Across collections, the Codex agent achieves an average MDL ratio of 0.83, but a pass rate of 74.16%, much lower than LIBRARIAN's rate of 90.67%. Similarly, the Claude agent reaches a higher pass rate of 82.50%, which is still lower than LIBRARIAN, but an MDL ratio of 1.07 which is more complex than the original collection. We present the full results of the agents in Appendix D, along with results based on newer models such as Claude Sonnet 4 and codex-mini. With Claude Code and Sonnet 4, agents achieve an MDL ratio of 0.77 and pass rate of 84.4%, outperforming codex-mini at an MDL ratio of 86.8 and pass rate of 82.0%.

MINICODE-Repositories Coding agents results on both small and large repositories are given in Table 2. During experiments, we found that the Claude Code agent provided superior plans for refactoring, while both the Claude Code and Codex agents performed satisfactorily at implementing a plan.

Despite using code agents with a human in the loop, existing code agents fail to produce effective refactorings. Often, the resulting refactored codebase bloats in size and complexity, particularly when using Claude Code for both planning and implementation. Given the poor performance of code agents on these repository-scale refactorings, we do not run the full sample-and-rerank pipeline. Supporting qualitative analysis in Section 8.4 discusses common failure modes of code agents on this task in repository-size refactorings.

8 Analysis

We further analyze parts of LIBRARIAN, consider alternative objective functions, analyze the use of clustering while constructing MINICODE-**CodeContests**, and common failure modes of code agents.

8.1 Can these libraries be reused to solve downstream programming problems?

To understand the practical value of our library extraction work for program synthesis, we augment the existing library learning algorithm, REGAL with sampling and MDL-reranking. In REGAL, the learned library is used to solve downstream holdout program synthesis problems, and a simpler clustering algorithm is used. For fair comparison with REGAL we hold the clustering algorithm constant (using the simpler REGAL approach), but sample $K = 5$ refactorings/libraries and rerank by MDL. The resulting system outperforms REGAL on holdout programming problems (Table 3).

These problems however are highly simple, which is representative of classic library learning work: Either drawing simple geometric designs (dataset Logo) or manipulating textual representations of dates (dataset Date). This helped motivate us to consider refactoring whole repositories.

Collection	Agent	MDL%	Pass%
datapipe	orig	100	100
	Cl-Cl	910	100
	Cl-Cx	220	100
state_machine	orig	100	100
	Cl-Cl	630	90
	Cl-Cx	190	100
config_schema	orig	100	100
	Cl-Cl	250	0
	Cl-Cx	210	100
cli_tools	orig	100	100
	Cl-Cl	400	100
	Cl-Cx	310	0
cli_form	orig	100	100
	Cl-Cl	380	100
	Cl-Cx	290	100

Collection		MDL%	Pass%
filesys_analyzer	orig	100	100
	Cl-Cl	160	100
query_language	orig	100	100
	Cl-Cl	140	1.0
knowledge_store	orig	100	100
	Cl-Cl	140	1.0
vm_emulator	orig	100	100
	Cl-Cl	160	100
finance_tracker	orig	100	100
	Cl-Cl	180	60
in_memory_db	orig	100	100
	Cl-Cl	150	100
text_editor	orig	100	100
	Cl-Cl	120	100

Table 2: The correctness (Pass %) and compression (MDL ratio as %) of the original and refactored code sources for the agent baselines on a subset of the small and large repository collections in MINICODE. Libraries with syntax or runtime errors receive a pass rate of 0.

Table 3: Solving downstream program synthesis tasks using learned libraries

Dataset	Model	Pass Rate
Logo	REGAL (gpt-3.5-turbo)	49.3% ±1.1
	LIBRARIAN (3.5-turbo)	69.9% ±0.9
Date	REGAL (gpt-3.5-turbo)	90.2% ±0.5
	LIBRARIAN (3.5-turbo)	94.7% ±0.7

8.2 What objective function should a library learner use?

We run our method on CodeContests using MDL, number of tokens, and cyclomatic complexity [25][4] as objective functions on 6 collections (Figure 3) While minimizing MDL also minimizes the other two objectives, the converse is not true. This suggests that MDL is a pareto-optimal loss among the three objectives in this experiment.

To confirm that the library does indeed expose shared abstractions, we calculate the average number of times that each library routine is used. Scaling the inference budget to $K = 8$ discovers better libraries, reusing each library function on average about 5 times.

In Appendix A.1, we find that MDL is also the objective that best correlates with human preferences in a small-scale human study.

8.3 Clustering Analysis: CodeContests

We analyze the coherence of the clusters underlying collections in MINICODE-**CodeContests**. In particular, we compare clustering based on o4-mini generated code source descriptions against task descriptions. Since task descriptions in competition coding problems are designed to hide the algorithmic approach needed to solve problem, we expect that clusters based on code source descriptions are more coherent.

[4] *Cyclomatic complexity* (CC) is a classic metric from the software engineering community that analyses the number of paths through a program's control flow, and unlike MDL is only loosely coupled with program size.

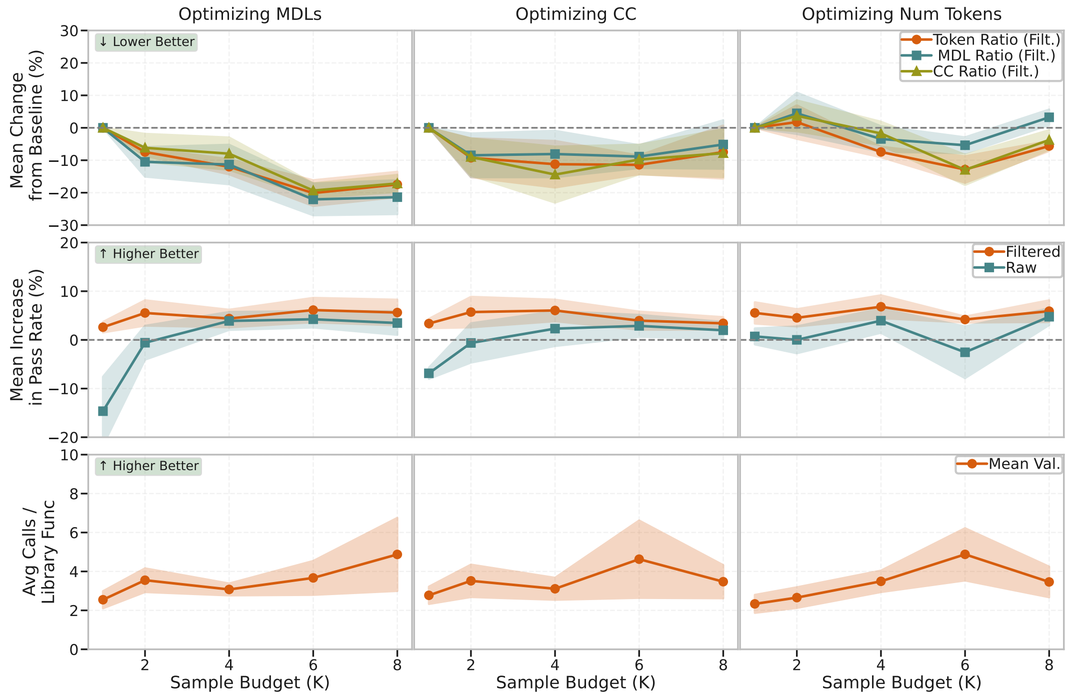

Figure 3: Comparing 3 different objective functions for refactoring (different columns) according to different downstream success metrics (different rows), as a function of refactoring budget (horizontal axes).The values are averaged over 6 collections of CodeContests problems. Row 1: Optimizing perplexity also incidentally optimizes cyclomatic complexity and token count, but that the converse is not true. Row 2: refactored programs pass more test cases, even more than the original code itself. Row 3: increasing the refactoring budget results in more reusable library subroutines (such subroutines are called more times on average). Filtered/Raw: Using/Not Using tests to filter samples.

We use two measures to evaluate the thematic coherence of collections: Good collections should group code sources with a (1) concentrated and (2) identifiable set of shared *conceptual tags*, which for CodeContests are provided as ground truth (`trees`, `graphs`, etc.).

Normalized Tag Instance Entropy This measures the concentration of tag *instances* within a collection, and is given by the entropy of the tag distribution for a given collection, normalized by the number of distinct tags in that collection. A lower normalized tag instance entropy (closer to 0) indicates higher thematic purity, meaning a small number of tag types are most prevalent.

Herfindahl-Hirschman Index (HHI) for Problem Presence This measures tag concentration across distinct code sources in a collection. A higher HHI signifies that the problems are collectively characterized by a smaller, more focused set of tags.

We provide the full formal definitions of both measures in Appendix B.

Figure 4 shows our clustering approach yields more thematically coherent clusters, evidenced by achieving lower entropy and higher HHI values across the entire tested range of N.

8.4 Common Failure Modes of Coding Agents

Failing to follow the plan. Refactoring large repositories requires editing code over a long time horizon. We instruct agents to plan hierarchically by first recording their plan in natural language, then implementing that plan. However, agents often ignore the plan, particularly with Codex as the implementation agent, potentially refusing to edit the code at all. We hypothesize that this behaviour is caused by the refactoring task itself – the original code source passes all tests, and certain agents are rewarded only for correctness rather than compression.

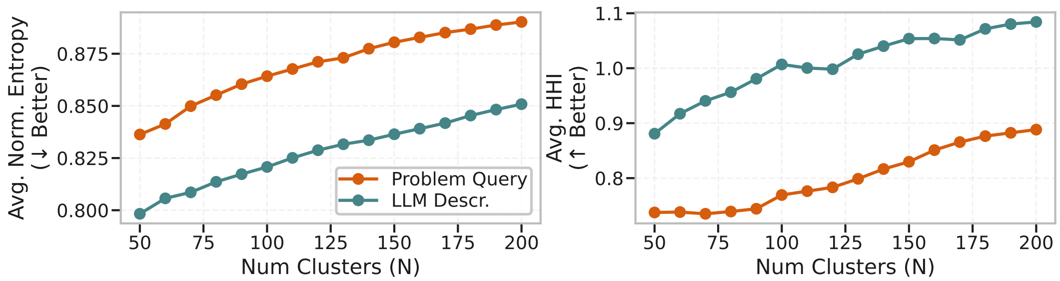

Figure 4: Clustering analysis of 4,596 Code Contest problems, comparing the thematic coherence of clusters formed using our proposed method versus REGAL-style clustering.

Failing to finish the task. For domains with a larger collections of code sources, such as CodeContests, agents may prematurely give up before refactoring all code sources. This happens despite explicit instructions to refactor all code. We attempt to fix this by making multiple calls to agents, allowing agents to resume the task. We anticipate that long-horizon instruction following capabilities will quickly improve with future models and agents.

Failing to use the library. When the implementation agent does follow the plan, it often fails to do so effectively. Claude Code, for example, may write a library, import it to the proper place, then rewrite the library rather than use it. We provide further examples in Appendix G.

9 Conclusion

We introduce a new benchmark MINICODE and method LIBRARIAN for compressing code sources through reusable abstractions. We highlight the challenges that modern models face when producing modular and maintainable code, then present an effective method for using LLMs to do this task in small-scale programs. By framing refactoring as both a design and compression task, our work opens new directions for building more general and scalable code understanding and generation systems. In particular, the structure of MINICODE lends itself well to reinforcement learning, where training would entail synthesizing collections of repositories to refactor then computing rewards based on compression.

Limitations Limitations of this work include the evaluation of synthetic repositories, the poor performance of code agents on refactoring, and the fact that compression may not be correlated with reuse. While our experiments on downstream programming problems partly address the question of reuse, investigating how to measure and encourage reuse for large-scale multi-repo library creation remains an open problem.

Ethics Statement

As with any coding benchmark, our work has the potential to improve automated coding. However, this has the potential to produce malicious code if left to run unchecked. Care should be taken to monitor inputs and outputs to the system.

Acknowledgements

KE was supported by NSF grant #2310350.

References

[1] Shane Tews. Inside tech's $2 trillion technical debt | american enterprise institute - aei.

[2] Catherine Wong, Kevin Ellis, Joshua B. Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, 2021.

[3] Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. Regal: refactoring programs to discover generalizable abstractions. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[4] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *PLDI*, 2021.

[5] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL), January 2023.

[6] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, 2013.

[7] Gabriel Grand, Li Siang Wong, Matthew Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code. *ArXiv*, abs/2310.19791, 2023.

[8] Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, volume 10, pages 639–646, 2010.

[9] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[10] Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations*, 2025.

[11] Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. Refactorbench: Evaluating stateful reasoning in language agents through code. In *The Thirteenth International Conference on Learning Representations*, 2025.

[12] Zhiruo Wang, Daniel Fried, and Graham Neubig. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks, 2024.

[13] Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. Ecco: Can we improve model-generated code efficiency without sacrificing functional correctness? *arXiv preprint arXiv:2407.14044*, 2024.

[14] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025.

[15] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 305–316, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.

[17] Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. *IJCAI*, 2017.

[18] Ray J Solomonoff. A formal theory of inductive inference. *Information and control*, 7(1):1–22, 1964.

[19] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[20] Joe H Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[21] OpenAI. Introducing o3 and o4 mini. `https://openai.com/index/introducing-o3-and-o4-mini/`, 2024. Accessed: 2025-05-13.

[22] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[23] Anthropic. Claude code: An agentic coding tool that lives in your terminal. `https://github.com/anthropics/claude-code`, 2025. An agentic coding tool that lives in your terminal, understands your codebase, and helps you code faster by executing routine tasks, explaining complex code, and handling git workflows - all through natural language commands.

[24] OpenAI. Codex: Lightweight coding agent that runs in your terminal. `https://github.com/openai/codex`, 2025. An open-source terminal-based coding assistant that provides ChatGPT-level reasoning with the ability to run code, manipulate files, and iterate - all under version control.

[25] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

A Algorithm

Algorithm 1 Refactoring Specialized Programs into a Joint Library

Require: Set of independent, specialized programs $P_{initial} = \{\rho_1, \rho_2, \ldots, \rho_n\}$
Require: Sample Budget K
Ensure: Joint library $\mathcal{L}_{final}$ and set of refactored programs P_{final}

$\mathcal{C} \leftarrow$ Cluster$(P_{initial})$
$\mathcal{L}_{final} \leftarrow \emptyset$, $P_{final} \leftarrow \emptyset$
for all cluster $c \in \mathcal{C}$ **do** ▷ Each cluster independently
  $T_C \leftarrow$ GroupIntoTuples(c) ▷ Get tuples for each cluster
  for all tuples $\tau \in T_C$ **do**
    $\{f_{retrieved}\} \leftarrow$ RetrieveRelevantFromLibrary$(\mathcal{L}, \tau)$
    $S \leftarrow \emptyset$
    for $i = 1$ to K **do** ▷ Sample k times
      $(\{f_{new,i}\}, \{\rho_i'\}) \leftarrow$ SAMPLE $(f_{retrieved}, \tau)$
      $S \leftarrow S \cup \{(\{f_{new,i}\}, \{\rho_i'\})\}$
    end for
    $(f_{best}, \{\rho_{best}'\}) \leftarrow$ RerankAndSelectBest$(S, \ell(\cdot))$ ▷ Rerank using objective
    $\mathcal{L}_{final} \cup \{f_{best}\}$
    $P_{final} \cup \{\rho_{best}'\}$
  end for
end for
return $\mathcal{L}_{final}, P_{final}$

A.1 Objective Functions

Human evaluation best aligns with the MDL objective function. We collect 19 CodeContests problems and a pair of refactorings for each, and have 4 human judges pick which they prefer. Each refactoring comprises three rewritten programs, plus the library generated by LIBRARIAN. Each pair of refactorings passed all testcases, and were chosen to include both the highest and lowest perplexity refactorings. Human judges are authors on this paper, but were 'blinded' to not know relative objective function scores for the candidate refactorings.

An MDL objective aligns best with human preferences (Figure 5), although number of tokens is also a reasonable proxy for human judgment. That token count and MDL yield different statistics furthermore implies that the lowest MDL program is not always the shortest program.

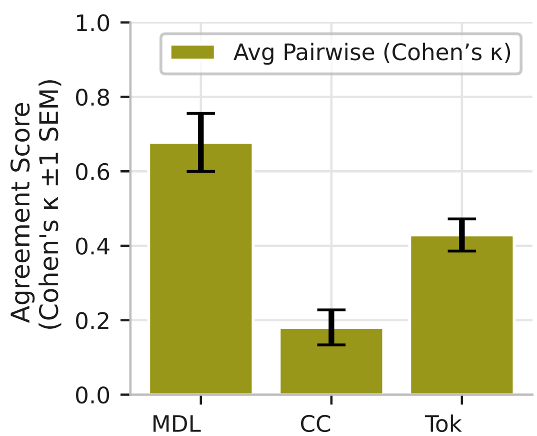

Figure 5: Human evaluation of different refactoring objectives. Humans and models give preferences to pairs of refactorings/libraries.

The instruction provided to human evaluators is as follows:

```


## 1. Materials Provided

You will be given a set of files for each example case:

* **`original_programs.py`**: This file contains a set of 3 distinct Python programs, each presented
    with its corresponding problem description/query. This represents the "before" state.
* **`v1.py`**: This file presents the first refactoring approach. It includes:
    * The 3 refactored versions of the original programs.
    * A "library" section (e.g., `codebank.py` or inline) containing helper functions. These helper
      functions might be retrieved from an existing common library or newly created during this
      refactoring.
    * Either the retrieved or the new helper function sections may be _empty_, in case no programs
      existed in the codebank at the time or if no need helper functions were created by the LLM.
* **`v2.py`**: This file presents the second, alternative refactoring approach. Similar to `
    refactoring_v1.py`, it includes:
    * The 3 refactored versions of the original programs (using a different strategy than v1).
    * A "library" section with its own set of helper functions.


**NOTE**:  both refactorings had accuracy at least as good as the original programs.

## 2. Your Task

Your primary task is to:

1.  **Review** the `original_programs.py` to understand the initial code and the problems being solved.
2.  **Analyze** both `refactoring_v1.py` and `refactoring_v2.py`. Pay close attention to how the
    original programs have been restructured and what functionalities have been extracted into their
    respective libraries.
3.  **Decide which refactoring (Version 1 or Version 2) you believe is "better,"** based on the
    evaluation criteria provided below (or your own criteria!).

## 3. Evaluation Criteria: What to Consider for Your Choice

When comparing `refactoring_v1.py` and `refactoring_v2.py`, please *consider* the following aspects to
    inform your choice. The "better" refactoring should ideally excel in these areas:

> Most importantly, make sure that the extracted functions are **actually reusable and not too specific
    .** If the main programs are short, the refactoring is not immediately "better"! Try to think
    whether the extracted functions could actually be used in a different program down the line.

* **Reusability of Helper Functions :**
    * **Generality:** Are the new helper functions general-purpose and potentially useful for *other,
      different* programs and problems beyond the three presented?
    * **Reuse:** How much were existing helper functions reused?
    * **Specificity:** Are the functions too specialized to the current set of problems, limiting their
      broader applicability? _Avoid functions that are essentially just the original program broken out
      into a "helper."_
    * Composability
* **Maintainability:**
    * Readability & Understandability
    * Ease of Modification
    * Separation of Concerns

## 4. What NOT to Focus On:

* **Comments:** Please disregard the presence or absence of comments in the code for this evaluation.
    These are superficially generated by LLMs in some occasions and could be added manually after with
     a single pass.
* **Minor Stylistic Differences:** Do not focus on trivial differences in variable naming or formatting
    , unless they significantly impact readability or understanding.

## 5. How to Provide Your Feedback

For each example case, please provide:

1.  **Your Preferred Version:** (e.g., "Version 1" or "Version 2")
```

Listing 1: Human Evaluation Instruction

B Collection Coherence Measures

We provide the full definitions of the collection coherence measures.

Normalized Tag Instance Entropy: This measures the concentration of tag *instances* within a collection C. Let p_i be the proportion of the i-th unique tag type among all tag instances in C, and

D_C be the number of distinct tag types in C. If $D_C > 1$, the normalized entropy H_N is defined as:

$$H_N = -\frac{\sum_{i=1}^{D_C} p_i \log_2 p_i}{\log_2 D_C} \tag{3}$$

If $D_C \leq 1$, then $H_N = 0$. Lower H_N (closer to 0) indicates higher thematic purity, meaning fewer tag types dominate the bulk of tag mentions.

Herfindahl-Hirschman Index (HHI) for Problem Presence: This measures tag concentration across distinct *problems* in a cluster C. Let s_t be the proportion of problems in C that include tag t (a problem contributes to s_t if t is one of its unique tags). A higher HHI signifies that the problems are collectively characterized by a smaller, more focused set of tags.

$$\text{HHI} = \sum_{t \in \text{Tags}(C)} s_t^2 \tag{4}$$

where $\text{Tags}(C)$ represents the set of unique tags present in cluster C.

C Benchmark Comparison

We compare our benchmark, MINICODE, to similar benchmarks in Table 4. We define creativity and design as the need to explore diverse solutions in order to find the best solution possible. For example, optimizing for program correctness alone does not require exploring a large solutions space, whereas optimizing a program for speed would. In the case of compressing large code sources, we must explore the large space of shared abstractions afforded by libraries in order to maximize compression.

Table 4: Comparison of Code Benchmarks

Benchmark	Creativity/Design	Scale
SWE-bench [9]	Low	Repository
Commit-0 [10]	Medium	Repository
RefactorBench [11]	Low	File
ECCO [13]	High	Function
KernelBench [14]	High	Function
MINICODE(Ours)	High	Multi-repository

D Full MINICODE Results

We present the full agent scores for the CodeContests split in Table 5. The results are given both for each cluster of code sources, as well as averaged across clusters.

Cluster	Agent	Tokens	CC	Pass %	MDL	MDL %
0	original	9088	95	80.3	11745.85	100.0
	sonnet 3.7	18114	176	87.0	15005.18	127.7
	sonnet 4	11121	138	80.3	9901.53	84.3
	codex-mini	9321	95	80.3	9990.74	85.1
1	original	12531	255	89.7	13431.86	100.0
	sonnet 3.7	10470	239	96.7	8933.65	66.5
	sonnet 4	11325	298	96.7	8214.42	61.2
	codex-mini	12762	255	89.7	11798.73	87.8
2	original	14087	376	89.0	15012.77	100.0
	sonnet 3.7	17345	429	91.3	13145.02	87.6
	sonnet 4	14270	356	93.0	10522.66	70.1
	codex-mini	14318	376	89.0	13273.81	88.4
3	original	14261	246	90.3	13348.82	100.0
	sonnet 3.7	20749	241	97.7	15859.02	118.8
	sonnet 4	13433	197	80.7	11937.04	89.4
	codex-mini	14495	246	90.3	11616.41	87.0
4	original	17693	336	80.7	14665.16	100.0
	sonnet 3.7	29860	358	100.0	20666.52	141.0
	sonnet 4	18684	352	82.0	12801.21	87.3
	codex-mini	17923	336	80.7	12902.09	88.0
5	original	12588	286	92.0	12790.11	100.0
	sonnet 3.7	10580	128	99.3	8435.12	65.9
	sonnet 4	10416	155	99.3	9167.85	71.7
	codex-mini	12819	286	92.0	11086.19	86.7
6	original	11020	131	54.3	13540.41	100.0
	sonnet 3.7	21747	502	88.0	19446.07	143.6
	sonnet 4	11177	143	57.3	10492.00	77.5
	codex-mini	11251	131	54.3	11651.65	86.1
7	original	12301	180	80.0	12393.73	100.0
	sonnet 3.7	16390	166	91.0	13371.59	107.9
	sonnet 4	11625	150	85.7	9304.25	75.1
	codex-mini	12534	180	80.0	10549.04	85.1
Avg	original	12946	238	82.0	13366.09	100.0
	sonnet 3.7	18157	280	93.9	14357.77	107.4
	sonnet 4	12756	224	84.4	10292.62	77.1
	codex-mini	13178	238	82.0	11608.58	86.8

Table 5: Comparison of the pass rate and compression metrics of the original code sources, Claude Sonnet 4 and codex-mini refactorings across CodeContests clusters.

We present the full repository-level results of MiniCode-repository in Tables 6 and 7, with o4-mini and Claude Sonnet 3.7 and 4.

Collection	Agent	LLoC	CC	MDL ratio	Pass rate
datapipe	original	474	116	1.0	1.0
	Cl-Cl	1084	341	9.1	1.0
	Cl-Cx	645	174	2.2	1.0
state_machine	original	619	222	1.0	1.0
	Cl-Cl	2271	735	6.3	0.9
	Cl-Cx	686	227	1.9	1.0
config_schema	original	949	284	1.0	1.0
	Cl-Cl	922	339	2.5	failed
	Cl-Cx	626	207	2.1	1.0
cli_tools	original	875	300	1.0	1.0
	Cl-Cl	2925	909	4.0	1.0
	Cl-Cx	5848	2378	3.1	failed
cli_form	original	352	174	1.0	1.0
	Cl-Cl	855	342	3.8	1.0
	Cl-Cx	1366	548	2.9	1.0

Table 6: Full results on MINICODE-repositories small, using Codex with o4-mini and Claude Code with Claude Sonnet 3.7.

Collection	Agent	LLoC	CC	MDL %	Pass %
command_line_task_manager	original	7964	2921	100	100
	sonnet 3.7	10387	3899	130	100
	sonnet 4	9515	3946	125	100
concurrent_task_scheduler	original	10350	3935	100	100
	sonnet 3.7	18653	7165	190	80
	sonnet 4	11491	4762	122	80
file_system_analyzer	original	3911	1338	100	100
	sonnet 3.7	6495	2218	160	100
	sonnet 4	5221	1793	130	100
in_memory_database	original	4565	1671	100	100
	sonnet 3.7	6667	2556	150	100
	sonnet 4	5617	2356	131	100
incremental_backup_system	original	4587	1482	100	100
	sonnet 3.7	5365	1788	130	failed
	sonnet 4	6620	2168	152	60
personal_finance_tracker	original	4306	1482	100	100
	sonnet 3.7	7443	2512	180	63
	sonnet 4	9212	3335	215	63
personal_knowledge_management	original	5341	1832	100	100
	sonnet 3.7	6357	2364	140	100
	sonnet 4	5575	2597	131	80
query_language_interpreter	original	5181	2105	100	100
	sonnet 3.7	7193	2966	140	100
	sonnet 4	7490	3076	142	100
text_editor	original	4324	1323	100	100
	sonnet 3.7	5792	1822	140	100
	sonnet 4	6017	2308	148	100
virtual_machine_emulator	original	6841	2283	100	100
	sonnet 3.7	10108	3580	160	100
	sonnet 4	9220	3303	137	100

Table 7: Full results on MINICODE-repositories large, comparing the original code sources with Claude Sonnet 3.7 and Sonnet 4.

E Data Generating Prompts

We provide the prompts for Librarian here.

````
I need ideas for Python libraries that can be implemented by language models. These libraries should:

1. Be implementable using only Python's standard library - no external dependencies
2. Have enough complexity to demonstrate sophisticated code design (10-20 functions/methods)
3. Include room for interpretation, so that different implementations can be unique while sharing core
      functionality
4. Have clear, practical utility that solves a real programming need
5. Be realistically implementable by an intelligent language model
6. Be testable with pytest
7. Include opportunities for different design approaches (functional vs OOP, etc.)

For each library, provide a description that outlines:
- The problem domain and core purpose
- Key required functionality (without being too prescriptive about implementation details)
- Potential use cases that demonstrate practical applications
- Suggested extension points where implementers could add their creative spin

Please generate several proposals in markdown, following this format:

```file:<library_name>/DESCRIPTION.md
# <Library Name>

## Purpose and Motivation
<3-5 sentences on what problem this library solves and why it's useful>

## Core Functionality
<Description of 4-6 high-level key features/capabilities without specifying exact implementation>
```

Be creative! Focus on domains where standard Python libraries provide enough building blocks but where
      a well-designed abstraction layer would add significant value.
````

Listing 2: Prompt to generate library descriptions

````

Consider a code repository designed to support the following task description:
```
{task_content}
```

Please list a couple dozen features that would be useful for the repository. Include the specified
      features as well as several others.

List the suggested feature names and descriptions in the following format:

1: <feature_1_name>: <feature_1_description>
2: <feature_2_name>: <feature_2_description>
3: <feature_3_name>: <feature_3_description>
...
30: <feature_30_name>: <feature_30_description>
"""

persona_prompt_template = """Consider the following features of a code repository:
{listed_features}

Think about several possibilities for what kind of person might use this code repository and what they
      might use it for. Please write several brief descriptions for the code repository in first person,
       formatted in markdown as follows:
```file:{library_name}/<persona_name>/TASK.md
# The Task

I am a <...> I want to be able to <...> This code repository <...>

# The Requirements

* `<function_name>` : <feature description>
* ...
```

Be creative! Write the task description in the style of the proposed persona. Be as exhaustive as
      possible in including the listed features in the task description's requirements.
````

Listing 3: Prompt to generate potential uses and features given a library description

````


I need you to implement a Python solution and COMPREHENSIVE suite of tests based on the following task files.
Your code must pass the tests provided.

{task_content}

CRITICAL FORMATTING INSTRUCTIONS:
1. You MUST format ALL code files exactly as shown below - no exceptions
2. Start each file with the markdown codeblock marker, followed by "file:" and the relative path
3. End each file with the closing markdown codeblock marker
4. Do not use any other format or markdown variations
5. For test files, do not put them in a subdirectory-- keep them in the outermost level.

For each source code file:

```file:<relative_file_path>
<file_content>
```

For each test file:

```file:<relative_file_path starting with test_>
<test_file_content>
```

IMPORTANT:
- The opening format must be exactly: ```file:path/to/file.py
- Do not add language indicators like ```python
- Do not add explanations between files
- Each file must be contained within its own codeblock with the precise format shown above
- The system parsing your response requires this exact format to function properly

EXAMPLE OUTPUT FORMAT:
```file:mymodule/mymodule.py
def example_function():
    return "This is a sample function"
```

```file:test_utils.py
import pytest
from mymodule.mymodule import example_function

def test_example_function():
    assert example_function() == "This is a sample function"
```

Begin your implementation now, following these formatting rules precisely.
````

Listing 4: Prompt to generate initial implementation

````

I need you to fix the implementation of the following code that is failing tests.

# Current Implementation:
{src_code_content}

# Test Files:
{test_content}

# Failed Tests:
{failed_test_details}

# Test Output (if available):
{test_output}

Please carefully analyze the errors and test failures. Pay special attention to:
1. The exact assertion failures or error messages
2. What the tests expect vs. what your current implementation provides
3. Any edge cases or special conditions you might have missed

Your task is to fix the implementation to make all tests pass. For each file that needs to be modified,
provide the content in the following format:

```file:<relative_file_path>
<file_content>
```

Where <relative_file_path> is the relative path to the file and <file_content> is the updated content of the file.
Focus on fixing the specific issues identified in the errors and failed tests while maintaining the overall structure of the code.

IMPORTANT: Make targeted changes to address the specific failing test cases. Make sure your implementation passes all test cases,
including any edge cases or special conditions mentioned in the tests. Be sure to output code in the specified format.
````

Listing 5: Prompt to fix code implementation, given pytest output.

F Refactoring examples of Librarian on Code Contests

F.1 Example 1

In code snippets 7, 6, 9, 8 one example of 2 refactoring versions. Specifically, the versions are both passing at least as many test cases as the original and they have the biggest difference in MDL among all the sample refactorings for that tuple. Sample + rerank filtering selected refactoring V2. You can observe that refactoring V1 introduces some problem specific functions like `build_max_beauty_perm()`, while refactoring V2 sticks to more generally useful functions.

```python
# ==== NEW HELPER FUNCTIONS ====
def compute_full_mask(i):
    """Return mask of all 1s of the bit-length of i."""
    return (1 << i.bit_length()) - 1

def build_max_beauty_perm(n):
    """Build permutation of 0..n maximizing sum of i^p[i]."""
    ans = [0] * (n + 1)
    used = set()
    for i in range(n, -1, -1):
        if i in used:
            continue
        mask = compute_full_mask(i)
        j = i ^ mask
        ans[i], ans[j] = j, i
        used.add(i)
        used.add(j)
    beauty = sum(i ^ ans[i] for i in range(n + 1))
    return ans, beauty

def solve_xor_sum(u, v):
    """
    Find shortest array whose xor is u and sum is v.
    Return list or None if impossible.
    """
    if u > v or (v - u) % 2:
        return None
    if u == v:
        return [] if u == 0 else [u]
    x = (v - u) // 2
    # try two elements
    if ((u + x) ^ x) == u:
        return [u + x, x]
    # fallback to three elements
    return [u, x, x]

def build_trie(keys):
    """
    Build a binary trie with counts for 30-bit numbers.
    Each node: [left_index, right_index, count].
    """
    tree = [[0, 0, 0]]
    for x in keys:
        now = 0
        tree[now][2] += 1
        for i in range(29, -1, -1):
            b = (x >> i) & 1
            if tree[now][b] == 0:
                tree[now][b] = len(tree)
                tree.append([0, 0, 0])
            now = tree[now][b]
            tree[now][2] += 1
    return tree

def trie_pop_min_xor(tree, x):
    """
    Pop one key from trie to minimize x^key and return that minimal xor.
    Decrements counts along the path.
    """
    now = 0
    res = 0
    for i in range(29, -1, -1):
        b = (x >> i) & 1
        nxt = tree[now][b]
        if nxt and tree[nxt][2] > 0:
            now = nxt
        else:
            now = tree[now][b ^ 1]
            res |= (1 << i)
        tree[now][2] -= 1
    return res
```

Listing 6: Version 1, New Helpers

```

# ########## PROGRAM: node_16:cc_python_16 ##########

from codebank import *

def main():
    import sys
    data = sys.stdin.readline()
    if not data:
        return
    n = int(data)
    perm, beauty = build_max_beauty_perm(n)
    print(beauty)
    print(*perm)

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_19:cc_python_19 ##########

from codebank import *

def main():
    import sys
    data = sys.stdin.readline
    n = int(data())
    A = list(map(int, data().split()))
    P = list(map(int, data().split()))
    trie = build_trie(P)
    O = [trie_pop_min_xor(trie, a) for a in A]
    print(*O)

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_25:cc_python_25 ##########

from codebank import *

def main():
    import sys
    u, v = map(int, sys.stdin.readline().split())
    res = solve_xor_sum(u, v)
    if res is None:
        print(-1)
    else:
        print(len(res))
        if res:
            print(*res)

if __name__ == "__main__":
    main()
```

Listing 7: Version 1, Refactored Programs

```
# ==== NEW HELPER FUNCTIONS ====
def compute_complement(i):
    return i ^ ((1 << i.bit_length()) - 1)

def trie_add(trie, x, max_bit):
    trie[0][2] += 1
    now = 0
    for i in range(max_bit, -1, -1):
        bit = (x >> i) & 1
        if trie[now][bit] == 0:
            trie[now][bit] = len(trie)
            trie.append([0, 0, 0])
        now = trie[now][bit]
        trie[now][2] += 1

def trie_find_min_xor(trie, x, max_bit):
    now = 0
    ans = 0
    for i in range(max_bit, -1, -1):
        bit = (x >> i) & 1
        if trie[now][bit] and trie[trie[now][bit]][2] > 0:
            now = trie[now][bit]
        else:
            now = trie[now][bit ^ 1]
            ans |= (1 << i)
        trie[now][2] -= 1
    return ans
```

Listing 8: Version 2, New Helpers

```
# ########## PROGRAM: node_16:cc_python_16 ##########

from codebank import *

def main():
    import sys
    input = sys.stdin.readline
    n = int(input())
    ans = [-1] * (n + 1)
    for i in range(n, -1, -1):
        if ans[i] == -1:
            z = compute_complement(i)
            ans[i] = z
            ans[z] = i
    m = sum(i ^ ans[i] for i in range(n + 1))
    print(m)
    print(*ans)

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_19:cc_python_19 ##########

from codebank import *

def main():
    import sys
    input = sys.stdin.readline
    n = int(input())
    A = list(map(int, input().split()))
    P = list(map(int, input().split()))
    max_bit = max(max(A, default=0), max(P, default=0)).bit_length() - 1
    trie = [[0, 0, 0]]
    for x in P:
        trie_add(trie, x, max_bit)
    res = [trie_find_min_xor(trie, x, max_bit) for x in A]
    print(*res)

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_25:cc_python_25 ##########

from codebank import *

def main():
    u, v = map(int, input().split())
    if u > v or ((v - u) & 1):
        print(-1)
    elif u == 0 and v == 0:
        print(0)
    elif u == v:
        print(1)
        print(u)
    else:
        w = (v - u) // 2
        if (w & u) == 0:
            d = u + w
            print(2)
            print(d, w)
        else:
            print(3)
            print(u, w, w)

if __name__ == "__main__":
    main()
```

Listing 9: Version 2, Refactored Programs

F.2 Example 2

In code snippets 11, 10, 13, 12 is another example of 2 refactorings where V1 was better according to LIBRARIAN. We can observe that V2 creates helper functions that are overly specific to the problem. You can see that refactoring V2 introduces overly specialized functions like `dijkstra_special()` or `compute_min_moves_opposite_parity()`. In comparison, refactoring V1 generates only general versions of these functions (e.g. `dijkstra()`).

```
# ==== NEW HELPER FUNCTIONS ====
def read_ints():
    return list(map(int, input().split()))

def build_adj_undirected(n, edges):
    adj = [[] for _ in range(n)]
    for u, v, w in edges:
        adj[u].append((v, w))
        adj[v].append((u, w))
    return adj

def dijkstra(adj, src):
    from heapq import heappush, heappop
    INF = 10**18
    n = len(adj)
    dist = [INF]*n
    parent = [-1]*n
    dist[src] = 0
    heap = [(0, src)]
    while heap:
        d, u = heappop(heap)
        if d > dist[u]:
            continue
        for v, w in adj[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                parent[v] = u
                heappush(heap, (nd, v))
    return dist, parent

def reconstruct_path(parent, dest):
    path = []
    u = dest
    while u != -1:
        path.append(u+1)
        u = parent[u]
    return path[::-1]

def multi_source_bfs(neighbors, sources):
    from collections import deque
    n = len(neighbors)
    dist = [-1]*n
    dq = deque()
    for u in sources:
        if dist[u] == -1:
            dist[u] = 0
            dq.append(u)
    while dq:
        u = dq.popleft()
        for v in neighbors[u]:
            if dist[v] == -1:
                dist[v] = dist[u] + 1
                dq.append(v)
    return dist
```

Listing 10: Version 1, New Helpers

```
# ########## PROGRAM: node_16:cc_python_16 ##########

from codebank import *

def main():
    import heapq
    n, m = read_ints()
    edges = [(u-1, v-1, w) for u, v, w in (read_ints() for _ in range(m))]
    adj = build_adj_undirected(n, edges)
    INF = 10**20
    dist = [INF]*n
    dist[0] = 0
    last_w = [0]*n
    heap = [(0, 0)]
    while heap:
        d, u = heapq.heappop(heap)
        if d > dist[u]:
            continue
        # record last edges
        for v, w in adj[u]:
            last_w[v] = w
        # expand two-edge moves
        for v, w1 in adj[u]:
            tw = last_w[v]
            for x, w2 in adj[v]:
                nd = d + (tw + w2)**2
                if nd < dist[x]:
                    dist[x] = nd
                    heapq.heappush(heap, (nd, x))
    out = []
    for x in dist:
        out.append(str(x if x < INF else -1))
    print(" ".join(out))

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_17:cc_python_17 ##########

from codebank import *

def main():
    n, m = read_ints()
    edges = [(u-1, v-1, w) for u, v, w in (read_ints() for _ in range(m))]
    adj = build_adj_undirected(n, edges)
    dist, parent = dijkstra(adj, 0)
    if dist[n-1] >= 10**18:
        print(-1)
    else:
        path = reconstruct_path(parent, n-1)
        print(*path)

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_19:cc_python_19 ##########

from codebank import *

def main():
    n = int(input())
    a = read_ints()
    # build reversed graph: for each move i->j, add edge j->i
    neighbors = [[] for _ in range(n)]
    for i, val in enumerate(a):
        for j in (i - val, i + val):
            if 0 <= j < n:
                neighbors[j].append(i)
    # BFS from all even and all odd positions separately
    even_sources = [i for i, val in enumerate(a) if val % 2 == 0]
    odd_sources  = [i for i, val in enumerate(a) if val % 2 == 1]
    dist_even = multi_source_bfs(neighbors, even_sources)
    dist_odd  = multi_source_bfs(neighbors, odd_sources)
    # for odd a[i], answer is dist to nearest even => dist_even; else dist_odd
    ans = [dist_even[i] if a[i] % 2 == 1 else dist_odd[i] for i in range(n)]
    print(*ans)

if __name__ == "__main__":
    main()
```

Listing 11: Version 1, Refactored Programs

```
#
# ==== NEW HELPER FUNCTIONS ====
def read_ints():
    return list(map(int, input().split()))

def build_undirected_weighted_graph(n, m):
    from collections import defaultdict
    adj = defaultdict(list)
    for _ in range(m):
        u, v, w = read_ints()
        u -= 1; v -= 1
        adj[u].append((v, w))
        adj[v].append((u, w))
    return adj

def dijkstra(adj, src, n):
    import heapq
    INF = 10**18
    dist = [INF]*n
    parent = [-1]*n
    visited = [False]*n
    dist[src] = 0
    heap = [(0, src)]
    while heap:
        d, u = heapq.heappop(heap)
        if visited[u]:
            continue
        visited[u] = True
        for v, w in adj.get(u, ()):
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                parent[v] = u
                heapq.heappush(heap, (nd, v))
    return dist, parent

def reconstruct_path(parent, dest):
    path = []
    while dest != -1:
        path.append(dest+1)
        dest = parent[dest]
    return path[::-1]

def dijkstra_special(e, n, src):
    import heapq
    INF = 10**18
    d = [INF]*n
    d[src] = 0
    heap = [(0, src)]
    while heap:
        cd, v = heapq.heappop(heap)
        if cd > d[v]:
            continue
        td = {}
        for u, w in e.get(v, ()):
            td[u] = w
        for u, w1 in td.items():
            for x, w2 in e.get(u, ()):
                cost = cd + (w1 + w2)**2
                if cost < d[x]:
                    d[x] = cost
                    heapq.heappush(heap, (cost, x))
    return d

def compute_min_moves_opposite_parity(a):
    from collections import deque
    n = len(a)
    go = [[] for _ in range(n)]
    ans = [-1]*n
    q = deque()
    for i, val in enumerate(a):
        for j in (i - val, i + val):
            if 0 <= j < n:
                if (a[j] % 2) != (val % 2):
                    ans[i] = 1
                    q.append(i)
                    break
                else:
                    go[j].append(i)
    while q:
        u = q.popleft()
        for v in go[u]:
            if ans[v] == -1:
                ans[v] = ans[u] + 1
                q.append(v)
    return ans
```

Listing 12: Version 2, New Helpers

```python
#
# ########## PROGRAM: node_16:cc_python_16 ##########

from codebank import *

def main():
    n, m = read_ints()
    e = {}
    for _ in range(m):
        u, v, w = read_ints()
        u -= 1; v -= 1
        e.setdefault(u, []).append((v, w))
        e.setdefault(v, []).append((u, w))
    d = dijkstra_special(e, n, 0)
    print(" ".join(str(-1 if x >= 10**18 else int(x)) for x in d))

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_17:cc_python_17 ##########

from codebank import *

def main():
    n, m = read_ints()
    adj = build_undirected_weighted_graph(n, m)
    dist, parent = dijkstra(adj, 0, n)
    if dist[n-1] >= 10**18:
        print(-1)
    else:
        path = reconstruct_path(parent, n-1)
        print(" ".join(map(str, path)))

if __name__ == "__main__":
    main()

# ########## PROGRAM: node_19:cc_python_19 ##########

from codebank import *

def main():
    n = int(input())
    a = read_ints()
    ans = compute_min_moves_opposite_parity(a)
    print(" ".join(map(str, ans)))

if __name__ == "__main__":
    main()
```

Listing 13: Version 2, Refactored Programs

G Code Agent Failure Modes

```python

from datapipe.core import (
    tumbling_window,
    sliding_window as _sliding_window,
    add_serializer,
    throttle_upstream as _throttle_upstream,
    watermark_event_time as _watermark_event_time,
    ...
)
...

def throttle_upstream(max_size):
    """
    Apply backpressure to slow data ingestion if downstream stages are overloaded.

    Args:
        max_size: maximum queue size or rate limit

    Returns:
        Decorator function
    """
    def decorator(func):
        from functools import wraps

        @wraps(func)
        def queue_wrapper(q, *args, **kwargs):
            try:
                size = q.qsize()
                if size > max_size:
                    import time
                    time.sleep(0.01)
            except Exception:
                pass
            return func(q, *args, **kwargs)

        return queue_wrapper

    return decorator

def watermark_event_time(events, allowed_lateness):
    """
    Assign event-time watermarks to handle late data correctly.

    Args:
        events: list of dicts with timestamp
        allowed_lateness: seconds of allowed lateness

    Returns:
        Events with watermark annotations
    """
    # Ensure we return the appropriate format with is_late field
    result = []
    for e in events:
        tagged = dict(e)
        max_ts = max(ev['timestamp'] for ev in events)
        watermark = max_ts - allowed_lateness
        tagged['watermark'] = watermark
        tagged['is_late'] = e['timestamp'] < watermark
        result.append(tagged)
    return result
...
```

Listing 14: Claude fails to use imports and instead re-implements the function.