Evaluating Large Language Models for Software Debloating

Abdel Rahman Ibrahim (919287448), Paiman Nejrabi (91953841), Sankalp Kashyap (924167993), Adil Bukhari (918713081), Devang Borkar (924164174)

Abstract

Software bloat—excessive or inefficient code that increases program size and reduces performance—continues to provide a considerable challenge in contemporary software development. This study investigates the efficacy of Large Language Models (LLMs) in eliminating software bloat without losing functionality. This study assesses how well LLMs work at identifying and removing software bloat. We evaluated the performance of various LLMs, including GPT-4, Claude 3.5 Sonnet, and DeepSeek Coder, on 76 Python code files from 26 repos through empirical evaluations. According to our data, while LLMs can considerably reduce the size of code (GPT-4o achieved the highest average reduction of 25.64%), this often comes at the cost of functional accuracy and execution time. Claude 3.5 sonnet was the most reliable of the models we tested, passing 97.11% of test cases, but it made more conservative reductions. Further prompt engineering experiments show that when functionality is prioritized in the instructions, the optimization results are improved. These results can help in finding the appropriate balance between the dilution of software and the software itself, which can help in the use of LLMs in the software design and development process.

1 Introduction

During the past decades, the size and complexity of software development have increased exponentially, and this has caused a massive issue called software bloat. Software bloat refers to the gradual build-up of inefficiencies or redundant code that decreases program performance and size without providing useful functionality. [1] It affects most aspects of software engineering, including execution efficiency, maintenance complexity, and security vulnerabilities. Bloat affects both large-scale business systems and resource-constrained mobile applications, and developers need to make efficient identification and elimination of such code a top priority.

Compiler optimizations, static analysis, and manual code inspections have comprised the standard software debloating procedures. Although these procedures hold promise, they often demand significant time investment, knowledge in their respective domains, and programmer skills. Additionally, human recognition and removal of bloat become less efficient and more error-prone as the size and complexity of codebases grow. Therefore, increasingly sophisticated automated tools must assist engineers in utilizing smaller, lighter codebases.

Large Language Models (LLMs), a recent artificial intelligence advancement, have been shown to be exceptionally good at comprehending and generating code in a variety of programming languages. LLMs can produce functionally equivalent code that can eliminate inefficiencies, comprehend code semantics, and recognize patterns. Though the potential applications of LLMs are tremendous, their potential to identify and solve software bloat while preserving the functionality of the original code remains inadequately researched. This lack of knowledge leads us to rigorously test the capabilities of LLMs to do code optimization.

1.1 Research Goals and Motivation

Our research evaluates the effectiveness of Large Language Models in detecting and eliminating software bloat while preserving functional correctness. Our objective is to furnish factual information and pragmatic insights that developers and researchers can utilize to enhance code quality and performance. Our objectives specifically encompass:

- **Evaluating debloating effectiveness:** We seek to measure LLMs' ability to reduce code size and complexity without compromising functionality.
- **Assessing functional correctness:** Through comprehensive testing, we aim to determine how reliably LLM-optimized code preserves the intended behavior and functionality of the original implementations.
- **Exploring prompt engineering strategies:** By experimenting with different prompting approaches, we intend to identify effective techniques for guiding LLMs toward optimal code transformations.

- **Comparing model architectures:** We aim to analyze performance differences among state-of-the-art LLMs to understand architectural strengths and limitations in code optimization tasks.

The motivation for this research goes beyond academic interest. Software bloat can adversely affect software development and maintenance in various ways—development expenses, user experience, resource utilization, and security stance.[2] We aim to utilize LLMs to aid with debloating initiatives, which can enable organizations to diminish maintenance burdens, increase performance, cut deployment sizes, and bolster code security. Moreover, effective debloating methods may enhance sustainable software practices by diminishing the energy expenditure linked to executing bloated code.

1.2 Research Questions

We have structured our investigation around four key research questions that systematically explore different aspects of using LLMs for software debloating:

RQ1: Can Large Language Models accurately detect and eliminate software bloat (as measured by reduction in code size and code correctness) from existing codebases while preserving code readability?

This research question examines the fundamental capability of LLMs to identify and remove unnecessary code while maintaining essential functionality. While we recognize that readability is an important aspect of code quality, our primary focus is on quantitative metrics of bloat reduction (lines of code, execution time) and functional correctness. Hence, we have evaluated readability on a small subset of the dataset mainly through automated metrics like Radon scores rather than subjective assessments, acknowledging that the relationship between code size and readability is complex and sometimes contradictory—something that can be researched in the future.

RQ2: How reliably do LLM-optimized code versions maintain functional correctness with their original bloated implementations across different programming paradigms as measured through test coverage and execution in open-source software repositories?

The second question addresses the critical concern of reliability in automated code transformation. We investigate whether LLMs can consistently produce functionally equivalent code across various programming paradigms and code structures. By systematically running comprehensive test suites on both original and optimized code, we evaluate whether the transformed code preserves the intended behavior under different inputs and conditions. This question is vital because even minor deviations in functionality could result in debloating efforts being counterproductive and potentially introducing subtle bugs or altering expected outputs.

RQ3: How does prompt engineering affect the ability of different LLMs to identify and optimize software bloat, and what impact does it have on maintaining code correctness?

This question explores how different prompting strategies influence the quality and reliability of LLM-generated code optimizations. We investigate whether specialized instructions, contextual information, and explicit constraints can guide models toward better debloating decisions. By comparing various prompt formulations, we aim to develop practical guidelines for developers seeking to leverage LLMs for code optimization tasks. This research question acknowledges that the effectiveness of LLMs largely depends on how they are instructed, and finding optimal prompting strategies could significantly enhance their utility in software engineering workflows.

RQ4: What are the comparative performance differences between different LLM architectures in generating optimized, unbloated code when measured against standardized metrics?

The concluding question investigates the impact of architectural variations among LLMs on their debloating efficacy. We want to find the best architectural techniques for optimizing certain types of code by testing a lot of different advanced models (GPT-4o, Claude 3.5 Sonnet, and DeepSeek Coder) with the same metrics and codebases. This comparison offers insights into the advantages and disadvantages of different model designs, assisting developers in selecting appropriate tools for their debloating requirements and informing future research in LLM development for code-related activities.

These research questions offer a thorough framework to investigate and understand the potential application of LLMs as software cleaning tools. They provide us with both theoretical and practical knowledge

applicable to the domain of automated code optimization.

2 Background

2.1 Literature Review

Existing research in software debloating has established a formal taxonomy of techniques based on input/output artifacts and debloating approaches, as per the SoK paper on Software Debloating.[3] Traditional methods have primarily been based on static analysis (code structure inspection without execution) and dynamic analysis (run-time behavior profiling), with CHISEL[4] and Mininode[5] as examples targeting source-to-source transformation, and others like BlankIt[6] operating at the binary level to enhance security by removing unused functions. Notably, machine learning approaches are not heavily employed in this domain, with only 7 of the 48 tools evaluated making use of ML approaches, and no tool employing Large Language Models (LLMs) to debloat code as of February 2025. Our work presents the innovation in our solution exploring the application of modern LLMs as a debloating approach.

Pertinent to addressing software bloat, the paper LLM Interactive Optimization of Open Source Python Libraries [7] offers a precursory perspective on the usage of language models on code optimization. While less focused on software bloat, it lays a substantial foundation in their approach utilizing LLMs, which inspired our methodologies. In it, Andreas Florath finds that an iterative dialogue with language models provides persistent optimization gains observed among standalone prompting and pair-programming with the model. Tested across Python libraries like Numpy and Pillow, Florath concluded that there appears to exist stronger potential in LLMs as a partner than as an autonomous solver, and strongly encourages human guidance: GPT-4o, the model of choice during his experiments, was successful in highlighting optimization patterns, suggesting various approaches, and recognizing edge cases and errors in its own code; however, when paired with an expert's observations rather than being given the responsibility of navigating the code, locating the missed opportunities for optimizations, then proposing a refactor, a 'pair-programming' approach led to the best results, including an instance that led to a 38x performance boost in one library's methods. Florath calls on researchers to explore the gaps that his paper did not regarding the usage of LLMs in other areas of software — bloat for example.

Expanding on the role of ML in debloating, the paper Leveraging Reinforcement Learning and Large Language Models for Code Optimization[8] introduces PerfRL, a framework that combines LLMs with reinforcement learning to optimize software code efficiently. By fine-tuning a lightweight CodeT5 model using specialized datasets and reinforcement learning from human feedback (RRHF), the study demonstrates that smaller models can outperform larger ones, such as CodeGen-16B, in generating optimized code while significantly reducing computational costs. Through a combination of greedy and random sampling techniques, the model learns to improve runtime performance while preserving functional correctness. The study evaluates its approach using metrics like Percent Optimized (%OPT) and Speedup (SP), showing a notable increase in performance over baseline models. Furthermore, by leveraging unit tests to validate optimizations, PerfRL aligns closely with our research goals of assessing LLMs' ability to detect and eliminate software bloat. These findings suggest that reinforcement learning can enhance LLM performance in code optimization, providing valuable insights into how different prompting strategies and model architectures may impact the effectiveness of bloat detection and reduction.

Looking at the effectiveness of the code while simultaneously ensuring the functionality is intact, Waghjale et al.'s ECCO framework [12] provides valuable insights for our software debloating approach. Although most studies focus on functional correctness alone, ECCO evaluates both correctness and efficiency metrics through two distinct paradigms: NL-based generation and history-based editing.

Their research revealed a persistent challenge: execution information and fine-tuning helped maintain code correctness, but no existing method reliably improved efficiency without compromising functionality. This fundamental trade-off directly informs our debloating work, though we take a different angle. We focus specifically on identifying and removing unnecessary code across diverse real-world GitHub projects rather than optimizing algorithms on competitive programming problems as ECCO does.

This paper by Fangzhou Wu et al. titled "Exploring the Limits of ChatGPT in Software Security Applications" provides a full analysis of the strengths and weaknesses of ChatGPT for the following software

security-related tasks: vulnerability detection, debugging, patching, and software debloating [9]. The authors indicate that while GPT-4 is performing reasonably well in code generation and analysis, a large codebase and complex dependencies present challenges. Moreover, ensuring proper functionality after modifications has become a significant challenge for this type of research. These limitations are particularly relevant to our project as we explore the role of LLMs in software debloating. Insights from this paper help justify key decisions in our methodology, including the selection of repositories with minimal interdependencies and the use of test cases to validate correctness after LLM-based modifications. This study focuses on the evaluation of ChatGPT in the security field and does not explore the impact of LLMs on software maintainability, readability, and performance beyond the security perspective. Our study aims to fill that gap by conducting systematic tests on multiple LLMs, including GPT-4o, Claude, and DeepSeek, on debloating Python test files and by determining their execution time, test coverage, and readability. Furthermore, we also intend to examine how the effectiveness of a prompt can be assessed by comparing models and their corresponding performance under varied prompts—an area of investigation not addressed in the original paper.

3 Data and Methods

3.1 Data Collection

3.1.1 Selection Methodology and Strategies

Our research required careful selection of appropriate software repositories to evaluate LLM effectiveness in code debloating. After initial exploration revealed many repositories had insufficient test coverage, we established several key criteria to guide our data collection process:

- **Programming Language Selection:** We settled on Python as our target language for several reasons. First, research by Wu et al. [9] suggests that LLMs process Python more effectively than lower-level languages. Second, Python's widespread use across diverse domains would make our findings more broadly applicable. Finally, Python's strong testing ecosystem would facilitate our evaluation of functional correctness.
- **Test Coverage Requirements:** We prioritized repositories with robust unit test suites, as these provided an objective mechanism to evaluate functional correctness before and after optimization. This criterion was essential for addressing our research questions regarding functional preservation during debloating.
- **Repository Independence:** We chose repositories with test files that were mostly independent of each other because LLMs have been known to have trouble handling complex dependencies between multiple files[9].

For systematic data collection, we utilized GitHub's search functionality with the query: `pytest.mark.parametrize language:Python`. We applied additional filters to identify repositories with at least 10 stars, ensuring a baseline level of community validation. After initial filtering, we conducted manual reviews of candidate repositories to confirm their suitability for our study.

3.1.2 Challenges Encountered

Our data collection process encountered several substantive challenges. A significant obstacle involved repositories with tests that appeared comprehensive but depended on complex setup procedures or external services. Test quality varied substantially across repositories. While some projects maintained exemplary test suites with clear assertions and comprehensive coverage, others implemented tests primarily for basic verification or relied heavily on mocks that inadequately verified functionality. We identified several repositories with high test counts but low actual coverage of code logic.

The persistent limitation of LLMs in understanding relationships between multiple files significantly constrained our selection. As documented by Wu et al. [9], even advanced LLMs struggle to track complex dependencies across file boundaries. This necessitated prioritizing modules that were relatively self-contained or had dependencies explicitly imported within the file, which excluded many otherwise suitable candidates.

Environmental setup presented additional complications. Certain repositories required specific hardware configurations, system libraries, or particular operating system versions. These technical constraints further reduced our pool of viable candidates.

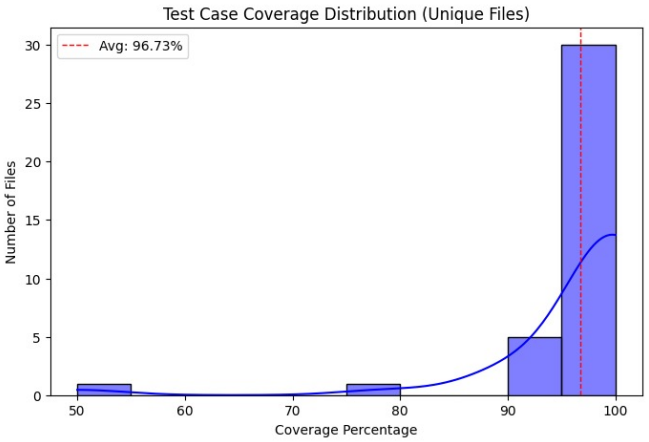

Figure 1: Test Coverage Distribution

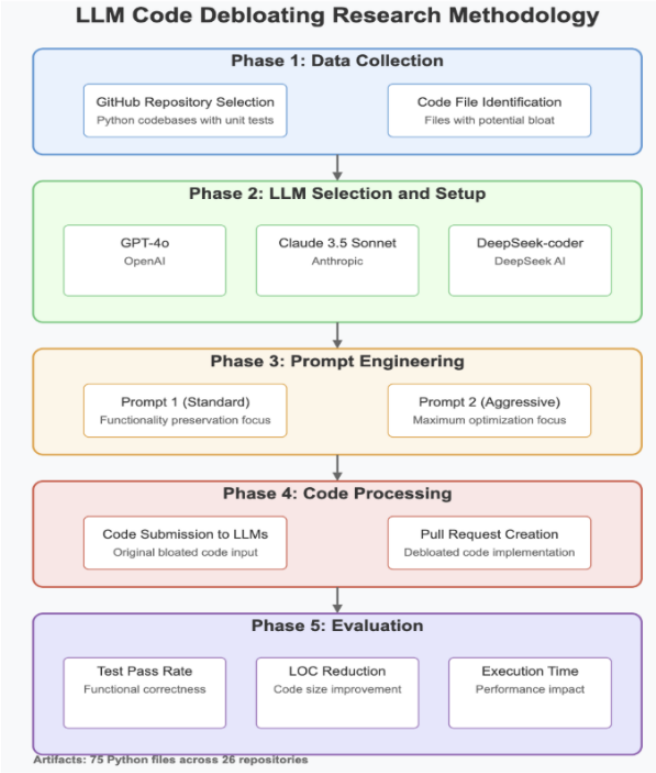

Figure 2: Five-Phase Methodology for LLM Code Debloating

3.1.3 Dataset Characteristics

Our final dataset comprised 26 repositories spanning diverse application domains—including web scraping frameworks (Scrapy), scientific computing libraries (Pymor), developer utilities, and specialized APIs. This domain diversity was essential for assessing whether LLMs' debloating capabilities generalize across different code types and programming patterns.

Within these repositories, we identified 75 Python files suitable for our experiments. We selected files exhibiting signs of potential bloat (excessive length, redundant operations, complex control structures) while maintaining adequate test coverage. The selected files varied significantly in size and complexity—ranging from 44 to 2,239 lines of code, with an average of 376 LOC.

The associated test suites reflected similar diversity. Across all target files, we counted 1,826 individual test cases, with some files having as few as 3 tests while others contained up to 148. The average of 38 test cases per file provided sufficient granularity to detect functional regressions that might result from debloating.

To ensure transparency and reproducibility, we maintained detailed documentation of all repositories, test files, evaluation metrics, and corresponding pull requests created throughout our experiments in a Google Sheet[10].

3.2 Experimental Methodology

Our research methodology followed a structured five-phase approach (Figure 2), designed to evaluate LLM capabilities systematically in code debloating:

Phase 1: Data Collection and Preprocessing

Beyond the selection criteria described above, we performed data preprocessing and cleaning to standardize the test files for consistent analysis. We developed a comprehensive Jupyter notebook to automate this process, ensuring uniform handling of test files across different repositories. This preprocessing step was crucial for maintaining consistency in our evaluation pipeline.

Table 1: Comparison of different approaches to code debloating prompts

Functionality Preservation Focus (Prompt 1):	**Simple Debloat (Prompt 2):**
A prompt emphasizing functionality preservation while optimizing code.	A simpler prompt to debloat code while maintaining core functionalities
Goal: You are an experienced software engineer. Please debloat the code in this file while maintaining its functional correctness. Simplify logic, remove redundancies, and optimize for readability and maintainability without introducing new bugs. **IMPORTANT:** 1. All rewritten code *must* remain within the file it originated from. 2. *No new files or services* may be introduced as part of the solution. 3. Adding helper methods within the file is allowed, but must not break functional correctness. 4. Do not modify OR remove comments, as they do not count as code. Imports also do not count as code. **Context:** Software bloat refers to unnecessary or inefficient CODE	"Debloat this file while maintaining functional correctness."

Phase 2: LLM Selection and Setup

We evaluated three state-of-the-art large language models, each utilizing different architectural approaches: GPT-4o (developed by OpenAI), Claude 3.5 Sonnet (from Anthropic), and DeepSeek-coder (by DeepSeek AI).

Our selection of these models was based on their documented performance in code-related tasks. GPT-4o and Claude 3.5 Sonnet were chosen due to their strong code generation and optimization track record, as demonstrated in previous studies.[11] DeepSeek-coder was included as a high-performance open-source alternative to compare how emerging open models perform against proprietary ones. While we experimented with some other open-source models, such as Llama 3.2 3B, we encountered limitations related to its context window, which made it difficult to process our code files effectively.

For each test file, we recorded several baseline metrics to ensure a comprehensive evaluation. These included the initial test case pass rate, execution runtime, lines of code (LOC), and test coverage percentage.

Phase 3: Prompt Engineering

We developed and tested two different prompting strategies to evaluate their impact on the effectiveness of debloating. We utilized GPT-4o to analyze and compare the effects of different prompts on an LLM.

Phase 4: Code Processing and Implementation

For each chosen repo, we carried out systematic processing. We created a fork of the repository and cloned it locally. Then, we checked that everything was working as it should. After that, we submitted the code to each LLM using one prompt variant, with GPT-4o being the only model that vetted both prompts. We used the optimized code made by the LLMs at the end. We created a unique debloating system written in Python to automate and standardize this. This system performs several tasks: it takes the path of a

test file as input, captures initial metrics (such as lines of code, test pass rates, and execution time) in an Excel sheet, interfaces with the selected LLMs using specified prompting strategies, logs comprehensive performance data, creates backups of the original code for comparison, and facilitates batch processing of multiple files. The system has various configurable options, allowing users to select their preferred LLM model (GPT-4o, Claude, or DeepSeek coder) along with their preferred prompt variant(s). This system uses the functionality preservation prompt (Prompt 1) by default; however, users can change this based on what they want to optimize the code for. All data collected during the experimentation is present in the main Google Sheet for evaluation.

Phase 5: Comprehensive Evaluation

We conducted a thorough comparative analysis of the original and debloated code versions using multiple evaluation criteria:

- **Test Case Pass Rate:** We measured the percentage of tests that passed before and after debloating to assess functional correctness preservation.
- **Execution Time Efficiency:** We compared test execution times before and after optimization to evaluate performance impact.
- **Code Size Reduction:** We calculated the percentage reduction in lines of code while acknowledging the limitations of this metric (particularly when models removed comments rather than actual redundant code).
- **Test Coverage:** We used the command `pytest --cov=package_name path_to_test_file` to compute test coverage percentages before and after modifications.

Our evaluation system automatically generated visualization charts to facilitate analysis of these metrics across different LLMs and prompting strategies. The system also automatically backed up original code files and logged comprehensive terminal output, enabling thorough assessment of the debloating effects.

To ensure consistency in performance evaluation, we conducted the majority of test executions on a standardized environment—a MacBook Pro M3 Pro with an 11-core processor. This standardization of hardware made execution time measurements less likely to be off by small amounts. This lets us separate the effect of code debloating on performance from variations caused by hardware.

3.3 Analysis Methods

We employed a comprehensive set of analytical techniques to interpret our experimental results:

Statistical Analysis: We calculated descriptive statistics for each performance metric across LLMs and prompt variations. The results provided a quantitative foundation for understanding general performance trends.

Comparative Performance Analysis: We conducted a systematic comparison of various LLMs across all metrics to tackle RQ4, highlighting the respective strengths and weaknesses of each model architecture in the context of code debloating tasks.

Trade-off Analysis: We analyzed the connections between various competing metrics, such as lines of code reduction and test pass rate, to uncover possible optimization trade-offs and establish the best debloating strategies for different scenarios.

Outlier Removal: Outlier removal using the IQR method is performed to eliminate anomalous values in Test Case Passing Ratio, Test Case Execution Time Ratio, and LOC Reduction Ratio', thereby making the dataset representative of typical LLM behavior in software debloating operations. It's important since it increases statistical accuracy, enhances visualization, and refines analysis to representative data, as shown by reduced variability and more central summary statistics upon elimination. While some data is sacrificed, this cleaning step supports reliable and meaningful conclusions about debloating efficacy.

Data Visualization: We developed detailed visualizations of our insights using the matplotlib and seaborn libraries to clearly convey performance differences and trends across various models, prompting strategies, and repository types.

By employing this systematic approach, we rigorously addressed each of our research questions, ensuring scientific integrity and reproducibility. Our methodology strikes a balance between automated processing

and thorough human evaluation, allowing us to analyze a wide range of files while maintaining high-quality assessments.

4 Results

This section reports the results of our investigation of three LLMs—Claude-3.5 Sonnet, GPT-4o, and DeepSeek—in their abilities to identify and remove software bloat from current codebases. It examines their efficacy in decreasing lines of code (LOC), maintaining functional correctness (through test passing ratios), and optimizing execution time, as well as investigating prompt engineering effects.

Table 2: LLM Debloating Performance Summary

Model	Sample Size	Test Passing Rate	LOC Reduction	Execution Time Change
Claude-3.5	40	97.11% ± 6.18%	24.41% ± 15.29%	-4.53% ± 12.91%
DeepSeek	35	96.43% ± 7.11%	22.21% ± 15.02%	0.57% ± 11.34%
GPT-4o	46	96.52% ± 7.62%	25.64% ± 17.90%	-1.77% ± 11.77%

4.1 RQ1: Can Large Language Models accurately detect and eliminate software bloat while preserving code readability?

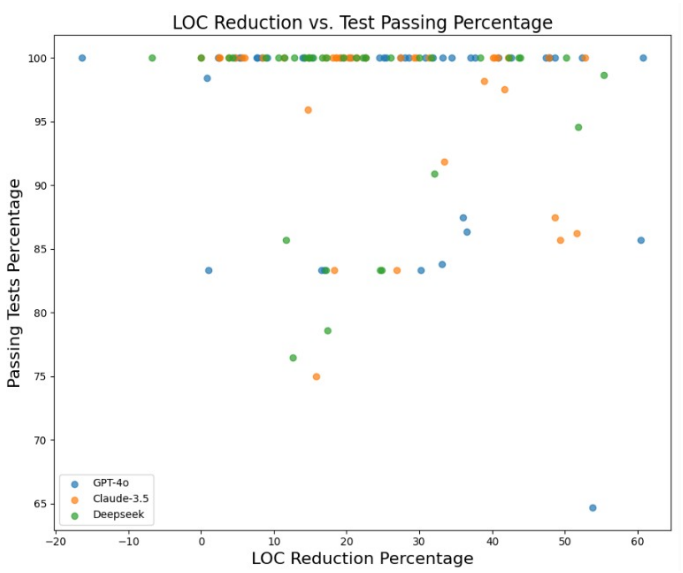

Figure 3: LOC Reduction Vs Test Passing Percentage

The analysis indicates that LLMs can optimize code efficiency without affecting functionality. Figure 3 shows how code size reduction and test success rates are related across models. Primary metrics are:

$$\text{Code Size Reduction \%} = \left(\frac{\text{LOC}_{\text{before}} - \text{LOC}_{\text{after}}}{\text{LOC}_{\text{before}}}\right) \times 100 \tag{1}$$

$$\text{Test Success \%} = \left(\frac{\text{Passing Tests}}{\text{Total Tests}}\right) \times 100 \tag{2}$$

$$\text{Execution Time Change \%} = \left(\frac{\text{Time}_{\text{after}} - \text{Time}_{\text{before}}}{\text{Time}_{\text{before}}}\right) \times 100 \tag{3}$$

Most notable is the dense cluster of results at the top of the plot showing nearly perfect test outcomes even at high percentages of reduction (up to 60%). Models like GPT-4o, Claude-3.5, and DeepSeek all demonstrate an impressive ability to eliminate redundant code while preserving functionality. Though extreme reductions occasionally show minimal dips in test performance, the overall trend demonstrates LLMs' ability to balance optimization and stability.

4.2 RQ2: How Reliably Do LLM-Optimized Code Versions Maintain Functional Correctness?

Table 3: Functional Correctness Assessment

Model	Sample Size	Test Passing Rate
GPT-4o	46	96.52% ± 7.62%

As shown in Table 3, LLM functionality correctness is generally strong, with test passing ratios around 96%. This high success rate underscores their real-world utility by significantly reducing the risk of introducing bugs during debloating. However, some test case failures still occur, indicating areas for improvement. We manually tried multi-shot prompting on a small subset of files and achieved 100% test pass rates, a promising direction for future work. As illustrated in Figure 1, most files selected for debloating had adequate test coverage, enabling high test passing ratios and demonstrating the reliability of the debloated files.

4.3 RQ3 & RQ4: Prompt Engineering Impact and Comparative Performance Differences

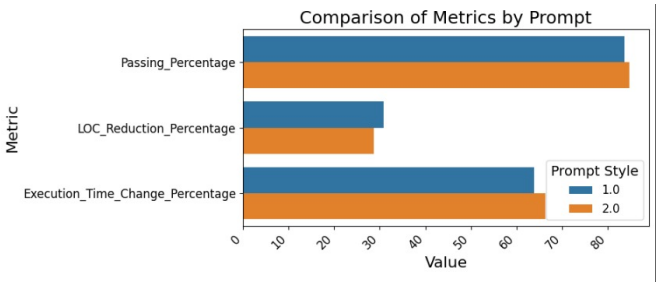

Figure 4: Prompt Engineering Impact

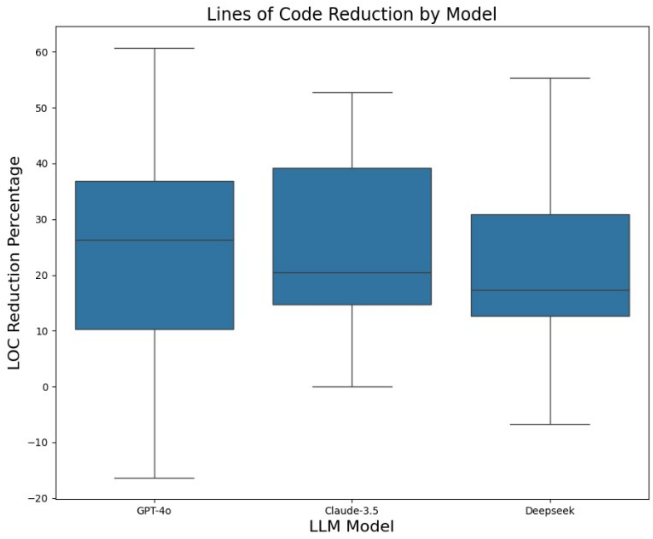

Figure 5: Comparative Performance

RQ3: Prompt engineering (detailed in Section 3.2 Phase 3) is beneficial to LLM performance. The detailed Prompt Style 1.0, with precise instructions and constraints, achieves marginally higher LOC reduction (30% vs. 25%) than the simpler Prompt Style 2.0. As shown in Figure 4, prompt engineering enhances GPT-4o's ability to detect and optimize software bloat without compromising correctness. However, the modest improvements suggest that prompt engineering doesn't dramatically impact model performance.

RQ4: Based on Table 2 and Figure 5, we identified key comparative differences across architectures:

1. **Test Passing Rate:** All models achieve high test passing rates above 96%, indicating they can produce functionally correct code after optimization, with variation within margin of error.
2. **Lines of Code Reduction:** GPT-4o achieves the highest average code reduction of 25.64%, followed by Claude-3.5 with 24.41%, while DeepSeek shows the most conservative reduction at 22.21%.
3. **Execution Time Impact:** Negative values indicate better (improved) execution times. Claude-3.5 improves execution time the most at -4.53%, followed by GPT-4o at -1.77%, while DeepSeek shows a slight slowdown of 0.57%.

5 Threats to Validity

5.1 Potential Limitations, Biases, and Challenges

- **Sample Size and Diversity:** The dataset of software projects used for debloating might not be sufficiently large or diverse to represent the wide range of real-world software scenarios. This limitation

could reduce the generalizability of the findings. We used a diverse range of software projects spanning different sizes and complexities. However, we acknowledge that using multiple programming languages would give a better picture of the LLM debloating landscape.

- **Model Selection Bias:** The study utilized specific LLMs—Claude-3.5, GPT-4o, and DeepSeek, that may not accurately reflect the broader spectrum of LLMs. The selection may skew the results if these models have certain strengths or weaknesses that tend the outcomes. To address this, several top LLMs were chosen based on their relevance and availability in the field. Their performance was compared with standardized tests to prevent bias and give a balanced evaluation.
- **Prompt Engineering Variability:** The effectiveness of the debloating process relied on prompt engineering, but the prompts might not have been perfectly optimized for all models and thus could have been biased for one model against others. To mitigate this, prompts were designed iteratively, incorporating feedback from initial tests, and the same set of prompts was applied consistently across only a single model to ensure fairness and reduce model-specific advantages.
- **Chosen Metrics:** Lines-of-code metrics, though valuable, might mask the deeper structural efficiencies gained or lost through optimization.

6 Discussion

The study evaluated three LLMs—GPT-4o, Claude-3.5, and DeepSeek—for their ability to detect and remove software bloat from Python codebases. Four research questions guided this analysis: (RQ1) effectiveness in reducing code size and runtime; (RQ2) reliability in maintaining functional correctness; (RQ3) the influence of prompt engineering; and (RQ4) comparative performance across model architectures.

6.1 RQ1: Effectiveness in Reducing Code Size and Runtime

GPT-4o demonstrated the greatest reduction in code size, emphasizing aggressive removal of redundant or repetitive constructs. This suggests GPT-4o primarily leverages surface-level optimizations, making it highly suitable for applications where minimal code footprint is prioritized. Claude-3.5, although slightly less aggressive, provided more cautious optimizations, possibly preserving key structural and semantic elements important for stability. DeepSeek, with the smallest reduction, likely employed simpler, heuristic-based optimizations rather than deeper semantic strategies.

Claude-3.5 notably led in runtime optimization, indicating targeted removal of performance-critical redundancies. GPT-4o's limited runtime improvement, despite its substantial LOC reductions, highlights that aggressive debloating alone does not guarantee enhanced efficiency. DeepSeek's slight degradation in runtime further emphasizes the importance of carefully targeted optimizations, as superficial or poorly directed debloating may inadvertently introduce inefficiencies.

This analysis underscores the necessity of deep semantic understanding in achieving effective runtime improvements. Models like Claude-3.5 appear better equipped to balance code reduction and execution efficiency, while GPT-4o and DeepSeek present opportunities to refine their optimization strategies beyond surface-level redundancies toward structurally meaningful code enhancements.

6.2 RQ2: Reliability in Maintaining Functional Correctness

Overall, the evaluated LLMs demonstrated strong reliability in maintaining functional correctness, consistently achieving high test-passing rates. Differences among Claude-3.5, GPT-4o, and DeepSeek were minimal, underscoring that modern LLMs are generally proficient at preserving intended functionality during code optimization tasks. Claude-3.5 displayed marginally lower variance, suggesting slightly more consistent behavior; however, the difference was minor and does not strongly support claims about significant methodological distinctions among models. GPT-4o and DeepSeek exhibited marginally greater variability, indicating a somewhat higher—but still limited—risk of occasional regressions.

Collectively, these findings highlight the practical utility of current LLMs in software optimization workflows, affirming their general capability to reliably maintain functional correctness. Claude-3.5's slight advantage may be beneficial in highly sensitive contexts, such as safety-critical or enterprise environments, where even small improvements in reliability can be meaningful.

6.3 RQ3: Impact of Prompt Engineering

Prompt engineering showed clear, though modest, trade-offs in optimization outcomes. The simpler prompt (Prompt 2) resulted in slightly higher test-passing rates and marginally better runtime improvements, suggesting that minimal prompting may encourage conservative optimizations that favor reliability and performance stability. Conversely, the detailed, engineered prompt (Prompt 1) yielded significantly greater reductions in lines of code, indicating that explicit constraints and detailed instructions encourage models like GPT-4o to pursue more aggressive structural simplifications.

These findings highlight an important practical consideration: developers must balance their prompting strategies according to specific optimization goals. While detailed prompts enable more substantial code reductions, simpler prompts may be preferable in contexts where maintaining strict functional correctness and runtime stability is the highest priority.

6.4 RQ4: Comparative Analysis of Model Architectures

The comparative analysis across the evaluated LLM architectures reveals distinct strengths and limitations that directly relate to their internal design and intended use cases.

GPT-4o excels in aggressive code minimization, emphasizing broad-scale removal of redundancies. This architectural characteristic likely stems from its tendency to identify repetitive or superficial patterns rather than deeper structural inefficiencies. However, GPT-4o's significant LOC reductions are not consistently reflected in execution time improvements, suggesting its optimizations may lack precision in targeting genuinely performance-critical segments of code. Thus, GPT-4o is well-suited for scenarios prioritizing minimal code size over runtime efficiency, such as embedded systems or resource-limited deployments.

Claude-3.5 offers a balanced optimization strategy, combining substantial LOC reductions with superior runtime improvements and consistent functional correctness. This suggests its underlying architecture employs deeper semantic reasoning, strategically identifying code redundancies that meaningfully influence both size and execution performance. Claude's approach positions it favorably for enterprise-level or performance-sensitive contexts, where reliability, execution efficiency, and moderate debloating collectively hold significant value.

DeepSeek demonstrates moderate performance, achieving lower code reductions and a slight deterioration in runtime efficiency. Its less aggressive optimizations indicate reliance on simpler, possibly heuristic-driven methods rather than in-depth structural or semantic understanding. However, a critical advantage of DeepSeek is its open-source nature, allowing organizations to host the model locally. This local deployment capability provides essential security advantages, addressing potential concerns regarding confidentiality and compliance, as organizations may prefer avoiding exposure of sensitive source code to external APIs.

Overall, these architectural distinctions suggest clear strategic choices for software engineering practitioners. Claude-3.5 best serves high-performance and reliability-critical scenarios, GPT-4o excels at aggressively minimizing code size, and DeepSeek, despite moderate optimization capabilities, offers a significant security and operational advantage through local deployment.

6.5 Comparison to Prior Studies

This study contributes to software debloating literature by exploring the novel use of Large Language Models (LLMs), an aspect that has not been addressed in conventional literature before. Traditional debloating methods, such as static and dynamic analysis methods exemplified by CHISEL[4], Mininode[5], and BlankIt[6], have focused on code-formatted or binary rewriting. Our research demonstrates the ability and practicability of LLMs like GPT-4o, Claude-3.5, and DeepSeek to autonomously optimize code bloat and runtime efficiency.

Our findings are largely in agreement with Florath's [7] GPT-4o optimization of Python libraries study, but our work greatly builds upon this endeavor through the report of important optimization obtained through independent, non-iterative interactions with LLMs. While Florath involved iterative human-model collaboration, our independent prompting techniques demonstrate that exceptional code optimizations can be achieved at the expense of drastic reductions and execution stability.

Also, our results regarding the significance of prompt engineering are corroborated by PerfRL's outcomes

on the utility of reinforcement learning in guiding LLM optimizations. Although reinforcement learning was not employed here directly, our results likewise stress that well-designed prompts significantly determine optimization depth and reliability, highlighting prompting as a practical means of achieving balanced debloating outcomes.

Further, our research confirms Waghjale et al.'s[12] ECCO model is capable of enhancing efficiency while maintaining correctness under control. By focusing specifically on diverse real-world GitHub repositories rather than controlled programming tasks, we place ECCO's theoretical findings into a more directly applicable software engineering context, providing tangible evidence of LLM's abilities and limitations.

6.6 Future Directions

Given the ever-changing landscape of LLM technologies and their nature to consistently improve on various coding benchmarks — some even showcasing up to 1.4x performance compared to previous-generation models[13] — the evaluation of newer models such as GPT 4.5, Claude 3.7-Sonnet in the context software debloat would prove necessary to remain competitive in addressing growing challenges in a quickly evolving space.

Similarly, further research across diverse programming languages, particularly languages with different runtime and memory management characteristics, would deepen the understanding of model applicability across contexts. As mentioned in Section 5, our evaluation strategy may overlook nuances in code structure. Future research should consider more comprehensive efficiency metrics such as execution profiling, cyclomatic complexity, or runtime resource utilization. Finally, a more extensive exploration into advanced prompt engineering strategies, such as iterative refinement or context-adaptive prompts, could further clarify how instruction specificity influences model behavior, potentially unlocking additional optimization capabilities. Prompting strategies, similar to those surveyed by Sahoo et al.[14] would include chain-of-thought (CoT) prompting, few-shot learning, role/persona assignment, and other formatting principles suggested by LLM providers.[15] [16]

7 Conclusion

This research demonstrates that LLM-based optimization provides meaningful opportunities for software debloating, with GPT-4o excelling in aggressive code reduction, Claude 3.5-Sonnet offering balanced, performance-conscious optimizations, and DeepSeek providing moderate, general-purpose reductions. Successful adoption of LLM-based debloating hinges on strategically aligning model choice with specific software engineering objectives, leveraging detailed prompt engineering, and ensuring rigorous validation through comprehensive testing. This integrated approach offers significant potential for developing software systems that are leaner, more efficient, and easier to maintain.

8 Team Membership and Attestation

Team members Abdel Ibrahim, Paiman Nejrabi, Sankalp Kashyap, Adil Bukhari, and Devang Borkar participated equally and efficiently.

Code & Data Availability Statement

Our code, which automates the debloating process, is available on GitHub. Additionally, all collected data and experimentation results recorded be accessed in this comprehensive google sheet [10].

References

[1] Mitchell, N., Sevitsky, G., & Coffman, T. (2006). *Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications*. Retrieved from https://www.researchgate.net/publication/221560136SoftwarebloatanalysisFinding.

[2] Muller, G., & University of South-Eastern Norway-NISE. (2020). *Exploration of the bloating of software* (pp. 1–3). Retrieved from https://www.gaudisite.nl/BloatingExploredPaper.pdf.

[3] Heo, K., Lee, W., Pashakhanloo, P., & Naik, M. (2018). *Effective program debloating via reinforcement learning*. In ACM SIGSAC Conference on Computer and Communications Security (CCS '18). Retrieved from https://chisel.cis.upenn.edu/papers/ccs18.pdf.

[4] Wang, S., Prakash, A., & Yin, H. (2018). *Alleystreet: Mitigating code-reuse attacks with control flow integrity*. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Retrieved from https://dl.acm.org/doi/10.5555/3277203.3277269.

[5] Koishybayev, I., & Kapravelos, A. (2020). *Mininode: Reducing the attack surface of Node.js applications*. In 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020). USENIX Association. Retrieved from https://www.usenix.org/conference/raid2020/presentation/koishybayev.

[6] Porter, C., Mururu, G., Barua, P., & Pande, S. (2020). *BlankIt library debloating: Getting what you want instead of cutting what you don't*. In 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020). Association for Computing Machinery. DOI: 10.1145/3385412.3386017.

[7] Florath, A. (2024). *LLM interactive optimization of open source Python libraries – Case studies and generalization*. arXiv preprint arXiv:2312.14949. Retrieved from https://arxiv.org/abs/2312.14949.

[8] Liu, J., Zhang, H., Chen, Y., & Wang, X. (2023). *Harnessing large language models for automated code optimization: Opportunities and challenges*. arXiv preprint arXiv:2312.05657. Retrieved from https://arxiv.org/abs/2312.05657.

[9] Wu, F., Zhang, Q., Bajaj, A. P., Bao, T., Zhang, N., Wang, R., & Xiao, C. (2023). *Exploring the limits of ChatGPT in software security applications*. arXiv preprint arXiv:2312.05275. Retrieved from https://arxiv.org/abs/2312.05275.

[10] [Dataset] "LLM Debloating Experiment Results," Google Sheets, 2024. Retrieved from Google Sheets.

[11] Stojanovic, M., Agre, G., & Ivanovic, M. (2024). *Large language models in software engineering: Capabilities, challenges, and opportunities*. Digital, 4(1), 85–105. Retrieved from https://www.mdpi.com/2673-6470/4/1/5.

[12] Waghjale, S., Veerendranath, V., Wang, Z. Z., & Fried, D. (2024). ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? arXiv preprint arXiv:2407.14044. Retrieved from https://arxiv.org/html/2407.14044v1.

[13] Anthropic. (2023). *Claude 3.7-Sonnet Announcement*. Retrieved from https://www.anthropic.com/news/claude-3-7-sonnet.

[14] Sahoo, S., Kumar, S., & Singh, R. (2024). *Exploring Prompting Strategies for Large Language Models*. arXiv preprint arXiv:2402.07927. Retrieved from https://arxiv.org/abs/2402.07927.

[15] OpenAI. (2024). *Prompt engineering guide*. Retrieved from https://platform.openai.com/docs/guides/prompt-engineering.

[16] Anthropic. (2024). *Prompt engineering overview*. Retrieved from https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/overview.

A Appendix:

Since this project is based on LLMs, we would like to mention that we extensively used them for data gathering and debloating process.

This appendix outlines potential directions for future research and development in software debloating, building on the current study's findings.

- **Measuring Readability Using Tools and Developer Input**: Current metrics overlook readability, which is critical for maintainability. Future work could integrate Radon for complexity analysis and developer surveys for qualitative feedback, ensuring debloated code remains functional.
- **Expanding Debloating to Diverse Programming Languages**: This work was limited to a few languages; now future work could target C++, Java, or Rust to identify improved results as these languages tend to have a more verbose nature compared to Python. Cross-language insights would broaden its applicability and robustness.
- **Exploring Multi-Shot Prompting for Enhanced Debloating**: Multi-shot prompting, could refine LLM outputs, this might improve debloating accuracy for complex codebases by offering richer context.