# MACEDON: Supporting Programmers with Real-Time Multi-Dimensional Code Evaluation and Optimization

Xuye Liu
University of Waterloo
Waterloo, Ontario, Canada
xuye.liu@uwaterloo.ca

Yuzhe You
University of Waterloo
Waterloo, Ontario, Canada
y28you@uwaterloo.ca

Xinrong Qiu
University of Waterloo
Waterloo, Ontario, Canada
a3qiu@uwaterloo.ca

Tengfei Ma
Stony Brook University
Stony Brook, New York, United States
tengfei.ma@stonybrook.edu

Jian Zhao
University of Waterloo
Waterloo, Ontario, Canada
jianzhao@uwaterloo.ca

## Abstract

Recent advancements in Large Language Models (LLMs) have led programmers to increasingly turn to them for code optimization and evaluation. However, programmers need to frequently switch between code evaluation and prompt authoring because there is a lack of understanding of the underlying code. Yet, current LLM-driven code assistants do not provide sufficient transparency to help programmers track their code based on the intended evaluation metrics, a crucial step before aligning these evaluations with their optimization goals. To address this gap, we adopted an iterative, user-centered design process by first conducting a formative study and a large-scale code analysis. Based on the findings, we then developed MACEDON, a system that supports multi-dimensional code evaluation in real time, direct code segment optimization, as well as shareable report displays. We evaluated MACEDON through a controlled lab study with 24 novice programmers and two real-world case studies. The results show that MACEDON significantly improved users' ability to identify code issues, apply effective optimizations, and understand their code's evolving state. Our findings suggest that multi-dimensional evaluation, combined with interactive, segment-specific guidance, empowers users to perform more structured and confident code optimization. The code for this paper can be found in <link-TBD>.

## CCS Concepts

• **Human-centered computing → User interface programming**; **Natural language interfaces**.

## Keywords

Code Evaluation and Optimization, Code Generation, Programming Interface, Large Language Models.

## 1 Introduction

The advent of Large Language Models (LLMs) has led to a paradigm shift in AI-driven code assistants [13, 20, 30, 61, 64] and brought transformative changes to programmers' workflow. Due to the lack of expertise, novice programmers heavily rely on LLM-driven code assistants in their workflow to generate or optimize sophisticated code from natural language (NL) prompts [23, 35, 53]. Instead of manually reviewing their code, programmers can now declaratively express their optimization goals to LLMs. However, this approach inadvertently overlooks the need for programmers to understand their code in depth or to conduct direction-oriented strategies for optimization, which could be challenging for novices when evaluating the quality of their generated code.

Recent research on programmers' interaction with LLM-driven code assistants has reflected these challenges for novice programmers. Specifically, uncertainty in generated code can lead to efficiency problems during the development process, as programmers are left to mentally anticipate possible outcomes until the code is generated, often requiring them to repeatedly refine their prompts until the desired result is achieved [6, 31, 49, 53]. While understanding the current state of code is crucial for crafting optimization prompts, a significant gap remains in supporting the complex process of forming optimization intentions. Therefore, our paper aims to explore designs that support the iterative process of LLM-driven code evaluation and optimization.

To address this gap, we first conducted a formative study involving six programmers with varying experience levels who regularly use LLM-based tools for code optimization. Our goal was to investigate how programmers assess their code and formulate optimization strategies in their existing workflows and the challenges they encounter. The study revealed a consistent need for *structured, multi-dimensional* code evaluation that surfaces aspects such as clarity, redundancy, documentation, time efficiency, and space efficiency. Participants noted that they often overlooked space usage or documentation quality unless specifically prompted. They also expressed the need to selectively optimize individual code segments without losing control over the broader codebase.

Building on insights from the formative study, we focus on exploring the practice and design of LLM-assisted code optimization and evaluation around the following research questions:

**RQ1 - Evaluation & Strategy.** *What evaluation dimensions and optimization strategies do programmers adopt when working with LLM-based assistants?*

**RQ2 - Tool Design & Effectiveness.** *How can we design effective tools that support multi-dimensional code optimization, and to what extent are they useful for real-world programming tasks?*

**RQ3 - Generalizability.** *Can such tools extend beyond novice programmers and remain effective in more diverse programming scenarios?*

To answer these RQs, we began by analyzing a large dataset of over 70,000 real-world code examples from the Performance-Improving Edits (PIE) dataset to identify five measurable dimensions for code optimization. This analysis provided a foundation for understanding how programmers evaluate and improve their code, answering **RQ1**. To explore effective designs for code optimization (**RQ2**), we developed MACEDON, a Visual Studio Code extension that supports multi-dimensional code evaluation and targeted, segment-specific suggestions. The tool was built based on three derived design goals: 1) helping users assess their code across the five dimensions using interpretable scores, 2) offering specific suggestions such as improving loop efficiency or renaming unclear variables, and 3) enabling users to apply changes consistently with minimal manual effort. To further examine whether this design is effective for real programming tasks (**RQ2**), we conducted a user study with 24 novice programmers. The results show that MACEDON helped participants evaluate their code easily and make more effective improvements. Specifically, compared to existing LLM code assistants, participants completed tasks more efficiently and produced code with higher expert ratings in clarity, maintainability, and performance. Furthermore, to explore whether MACEDON is useful beyond novice programmers (**RQ3**), we conducted two case studies in which participants applied the tool to real programming tasks. The case studies show that MACEDON supported users' goals, helped them organize their codebase, and remained effective in broader programming contexts.

In summary, our contribution is threefold:

- A formative study and a comprehensive data analysis of optimization behaviors and needs when using LLMs for code evaluation and optimization.
- An interactive tool, MACEDON, developed as a Visual Studio Code Extension, that supports segment-specific code optimizations in multiple dimensions, minimizing rework and improving efficiency.
- A user study and two case studies for assessing MACEDON, demonstrating its effectiveness in improving code quality and user experience compared to conventional LLM-based assistants.

## 2 Related Work

### 2.1 Generating and Optimizing Code with LLMs

Recent programming workflows have shifted towards leveraging LLMs for both generating and optimizing code. LLMs like GPT [1], Codex [13], CodeGen [40], and InCoder [19] can interpret natural language instructions and produce corresponding code snippets [17, 42, 46]. These models allow developers to specify high-level generation and optimization goals through natural language prompts, reducing the time spent on manual adjustments. For example, tools like GitHub Copilot utilize LLMs to assist developers in code completion and optimization by understanding the surrounding code context and generating suggestions [6, 39].

Other LLM-based frameworks have also been developed to address different challenges in code generation and optimization. For instance, ClarifyGPT [36] enhances the code generation process by identifying ambiguities in user prompts and seeking clarifications, ensuring that generated code aligns closely with user intentions. Similarly, the use of reinforcement learning in conjunction with LLMs has enabled models to adaptively improve code based on feedback from test cases, further refining the generated outputs [15]. CodePlan [5] addresses the challenge of complex repository-level tasks by using a task-agnostic, neuro-symbolic framework that frames coding as a planning problem, synthesizing a multi-step chain of edits through dependency analysis, change impact analysis, and adaptive planning with neural LLMs. Additionally, methodologies combining program analysis with LLMs, such as leveraging GPT-4 for identifying inconsistencies between code comments and their implementation, have shown success in automating code documentation maintenance and reducing developer effort [63]. SBLLM [21] combines LLMs with search techniques for iterative code optimization, using representative sample selection, adaptive pattern retrieval, and genetic operator-inspired prompting to achieve code efficiency improvements. CoLadder [58] provides a hierarchical structure for decomposing programming tasks, allowing programmers to better align their problem-solving intentions with LLM-generated code, thus improving the optimization and modification of code across various abstraction levels.

While prior research has explored the use of LLMs for coding tasks, as well as frameworks that address ambiguity in prompts and complex repository-level tasks, our work focuses on the iterative nature of the code evaluation and optimization process in natural language programming. Unlike search-based optimization [21] or task decomposition [58], our approach emphasizes providing programmers with real-time insights into their code's status through a structured interface that facilitates direct, segment-specific optimizations. MACEDON uniquely combines visualization of code state with targeted recommendations, enabling programmers to efficiently track and improve code quality through a seamless integration into their workflow. Additionally, our work fills a crucial gap in supporting programmers' understanding of their code before crafting optimization prompts, thereby offering a more comprehensive solution for iterative LLM-driven code optimization.

### 2.2 Evaluation of LLM-based Code Assistants

Evaluating the correctness and quality of code generated by LLMs is crucial for understanding their effectiveness in software development tasks. Traditionally, benchmarks like HUMANEVAL [13] have been used to measure functional correctness by running predefined test cases against generated Python functions. Similarly, MBPP [4] provides a set of crowd-sourced programming problems with corresponding test cases, allowing for a broader evaluation of LLMs' code generation capabilities. Expanding this to multiple languages, MultiPL-E [11] and HumanEval-X [65] translate these benchmarks to multiple other coding languages, enabling cross-language evaluations for models like Codex and CodeGen. For competitive programming, AlphaCode [30] uses complex challenges from Codeforces to test problem-solving capabilities, while Spider [59] assesses text-to-SQL conversion tasks. These benchmarks focus on determining whether the generated code meets specific functional requirements. EvalPlus [32] addresses the limitations of traditional benchmarks by generating a larger set of test cases through automatic test input generation, significantly expanding the scope of testing. Beyond isolated function synthesis, other benchmarks such as SWE-bench [25] focus on real-world software engineering tasks, where LLMs must resolve issues in complex codebases involving multiple files and components. SWE-bench evaluates LLMs by providing them with GitHub issues and corresponding codebases, requiring models to generate code patches that resolve the issues while passing the associated test cases.

Our work differs from these approaches by emphasizing the integration of real-time feedback and iterative evaluation during the code optimization process. While traditional benchmarks focus on assessing functional correctness through static test cases, our system, MACEDON, offers a dynamic, user-centered approach, allowing programmers to evaluate their code's state across multiple dimensions and refine it interactively with the support of LLM-driven recommendations. This approach bridges the gap between one-off code generation and the ongoing process of code refinement, offering a more practical solution for real-world programming tasks.

## 2.3 Traditional Code Analysis and Optimization Tools

Traditional approaches to code improvement have evolved through systematic analysis and transformation methodologies, encompassing both performance optimization and code refactoring techniques. Performance analysis tools like HPCToolkit [2] and Speedoo [14] collect runtime data to identify bottlenecks and prioritize optimization opportunities, while automatic performance regression detection systems [27] help maintain code efficiency over time. These tools provide valuable insights into execution characteristics but operate as separate analysis phases, requiring programmers to compile, execute, and analyze their code outside of the development workflow. Similarly, static analysis and refactoring tools, including ESLint [60], JDeodorant [52], and IDE-integrated capabilities in IntelliJ [22], offer automated code transformations and style enforcement based on predefined rules and patterns. While these tools excel at maintaining code consistency and applying well-established refactoring patterns [37, 41], they lack the contextual understanding and adaptive guidance needed for complex, domain-specific optimization tasks that require understanding of programmer intent and code semantics.

Recent advances in machine learning for code analysis have introduced new capabilities for automated code improvement. Neural program repair systems [54] and learning-based performance optimization [48] demonstrate AI-driven code enhancement. Mixed-initiative IDE research like Grounded Copilot [6] and in-IDE code generation [56] explore AI assistance in development environments. These approaches focus on code generation rather than evaluation and optimization workflows. Programmers still need real-time insights into their code's state across multiple dimensions. The challenge remains in providing programmers with real-time insights into their code's current state across multiple dimensions before they can effectively craft optimization strategies, as highlighted by quantitative assessments of development techniques [43].

Our work addresses these gaps by combining real-time, multi-dimensional code evaluation with interactive optimization. Unlike performance tools that require separate analysis phases, MACEDON provides immediate feedback during programming. While static refactoring tools use predefined rules, MACEDON uses LLM capabilities for context-aware optimizations. This integration of evaluation and optimization in one interface advances beyond tools that treat these as separate processes.

## 3 Design Process & Goals

We conducted an iterative user-centered design process to create MACEDON. The design process included three key stages: 1) *Understanding & Ideation*—involving an interview study with experienced programmers to uncover challenges and strategies in code optimization using LLM-driven tools; 2) *Prototype & Walkthrough*—the design and development of MACEDON, informed by the insights gained, followed by a cognitive walkthrough for feedback and iterative refinements; 3) *Deploy & Evaluate*—a user study to assess how programmers interact with the system and its perceived usefulness in streamlining code optimization. In this section, we describe the first stage of our design process, outlining the strategies and design goals that guided the development of MACEDON.

## 3.1 Interview Process

We recruited six participants (4 males, 2 females; ages $20-27, M = 23.5, SD = 1.2$) with different levels of programming experiences through purposive sampling [16] for our interviews. Our goal was to engage participants who were both experienced in programming and familiar with LLM-driven code optimization tools. During the recruitment process, we conducted a pre-test survey to screen for eligibility, measuring programming experience on a 5-point scale [1: very inexperienced; 5: very experienced], years of programming experience, and self-reported familiarity with code optimization tools driven by LLMs. Three participants are novice programmers with only around one year of programming experience while the other two have at least five years.

They were all familiar with programming (score $M = 4.17, SD = 0.41$), and regularly used LLM for code optimization purpose ($M = 7.5, SD = 2.10$ times/week). Participants gave consent and were compensated 20 CAD for a 60-minute study session.

Before the study, participants were asked to share their recent examples of ChatGPT usage for optimizing the code to nudge them to reflect their optimization strategies with LLM-code optimization tools. During the session, we interviewed participants to explore their challenges in understanding and evaluating the state of their code, translating them to optimize the code further, and their strategies for addressing these challenges and their needs. All interviews were audio-recorded and transcribed. We then performed thematic analysis [50] using both inductive and deductive approaches. Based on our analysis, we identified and categorized key themes and strategies employed by participants. Any disagreements were resolved through discussion, leading to final themes after a second iteration of analysis.

## 3.2 Interview Results

Here, we present our findings on the workflows that participants adopted during the code optimization process and the challenges they encountered.

*3.2.1 Multi-Dimensional Code Evaluation.* We observed that participants' approach to optimizing code involved two key aspects. First, programmers needed to assess the current state of their code across multiple dimensions, such as clarity, efficiency, and readability, which was crucial to determine *"where the code falls short and what needs to be improved."* -P2 Second, they had to explore how to implement optimizations in a way that addressed those specific shortcomings. However, participants encountered challenges with the lack of transparency in optimization recommendations, which made it difficult to understand how certain changes impacted their code. P4 noted, *"It's hard to know if the optimization is improving performance or just adding complexity"*. To alleviate cognitive load, every participant adopted a similar strategy to break down the optimization process into smaller, manageable steps.

Six participants suggested that having a structured approach to multiple dimensions would make the optimization process more efficient. P4 emphasized, *"I usually start with time efficiency, then clarity and documentation to ensure everything is readable and well-organized."* Additionally, two participants mentioned that they would easily overlook space usage and redundant code, which could affect others' ability to read the code efficiently. For example, P2 noted, *"I don't always think about space efficiency unless flagged."* P6 added, *"It's only after someone struggles with reading my code that I realize how important reducing redundancy is."* Thus, a multi-dimensional code evaluation approach is needed to optimize code in a systematic and organized manner.

*3.2.2 Facilitating Direct Code Segment Optimization.* Participants expressed challenges in managing large amounts of code, leading to a sense of control loss over the optimization process, where participants expressed the desire to *"optimize the code dimension by dimension"* -P2. Some (4/6) participants expressed frustration with the system applying changes across the entire code at once. Participants preferred direct manipulation of individual code segments based on specific priorities, such as time performance or clarity improvements. P5 explained, *"I prefer to fix one section at a time; it helps me see exactly what's being changed and why."* Participants (4/6) reported frequently using an alternative strategy where they optimized self-contained code segments independently before integrating them into the broader codebase. P4 noted, *"I prefer optimizing each part separately and then merging them so I can see the overall improvements gradually."* Another common approach involved selecting and optimizing specific segments of code based on previous runs, enabling programmers to refine code in targeted areas without affecting other sections. However, P5 also outlined this tedious process of preserving and comparing the newly optimized code and original code, *"I have to keep track of which parts are optimized and ensure they don't conflict with the rest of the code."*

*3.2.3 Real-Time Feedback for Iterative Code Optimization.* All participants expressed frustration with the constant switching between evaluating their code and applying optimizations, which often led to cognitive overload. P2 shared, *"Sometimes I need to stop and evaluate if the optimization worked; it breaks my flow and makes the whole process slower."* P5 agreed and mentioned that it was *"frustrating to go back and forth between evaluation and deciding whether to make changes."* However, we observed that participants preferred real-time feedback during the process, which helps them quickly verify *"whether the code change is accurate"* -P4. Some participants even described the benefits of iterative code optimization for each small code segment. P4 noted, *"I often focus on the areas that require the most attention first and do iterative prompting until it works."* The iterative method of refining individual sections allowed participants to remain in control of the process, avoiding broad changes that might affect the entire codebase and ensuring each adjustment was aligned with their optimization goals.

## 3.3 Design Guidelines

Based on the interview results, we derived the following design guidelines to drive the development of MACEDON.

***DG1: Providing Multi-Dimensional Feedback for Code Evaluation***. A lack of comprehensive evaluation across different metrics can prevent programmers from fully understanding the state of their code. The system should offer multi-dimensional feedback, such as performance, readability, and clarity, to help users better understand the strengths and weaknesses of their code. This evaluation should be presented in a structured and organized manner, allowing programmers to externalize their thought processes and refine specific areas of the code based on the feedback. Flexible feedback options should reflect the various dimensions programmers need for code optimization.

***DG2: Facilitating Direct Code Segment Optimization***. Programmers often feel a loss of control when working with large amounts of code that require detailed optimization. The system should support direct manipulation of specific code segments, enabling users to select and modify areas for improvement based on their priorities. It should also allow programmers to reorganize or refine their code incrementally, providing the ability to test and apply optimization recommendations to one segment at a time, instead of overwhelming them with global code changes.

***DG3: Integrating Real-Time and Interative Code Evaluation with Optimization Suggestions***. Programmers often face cognitive overload due to the constant need to switch between code evaluation and applying optimizations. The system should integrate real-time feedback with iterative suggestions, allowing programmers to continuously assess their code and apply optimizations without disrupting their workflow. By providing timely, contextual feedback during the optimization process, the system can guide users in refining their code in a smooth, iterative manner.

## 4 Define Code Evaluation Method

The insights of our formative study led to the investigation of our **RQ1** regarding evaluation dimensions and optimization strategies that programmers normally use. As indicated by DG1, we decided to evaluate the C++ code on five key metrics: redundancy, documentation, clarity, time efficiency, and space efficiency.

## 4.1 Data Collection

We utilize the Performance-Improving Edits (PIE) dataset [48] containing human-programmer optimizations from competitive programming tasks in CodeNet [44]. We selected C++ as our target language due to its prevalence in performance-critical applications and compatibility with benchmarking tools like the Gem5 simulator. C++ submissions typically emphasize fine-grained optimizations, making them ideal for analyzing time and space efficiency metrics.

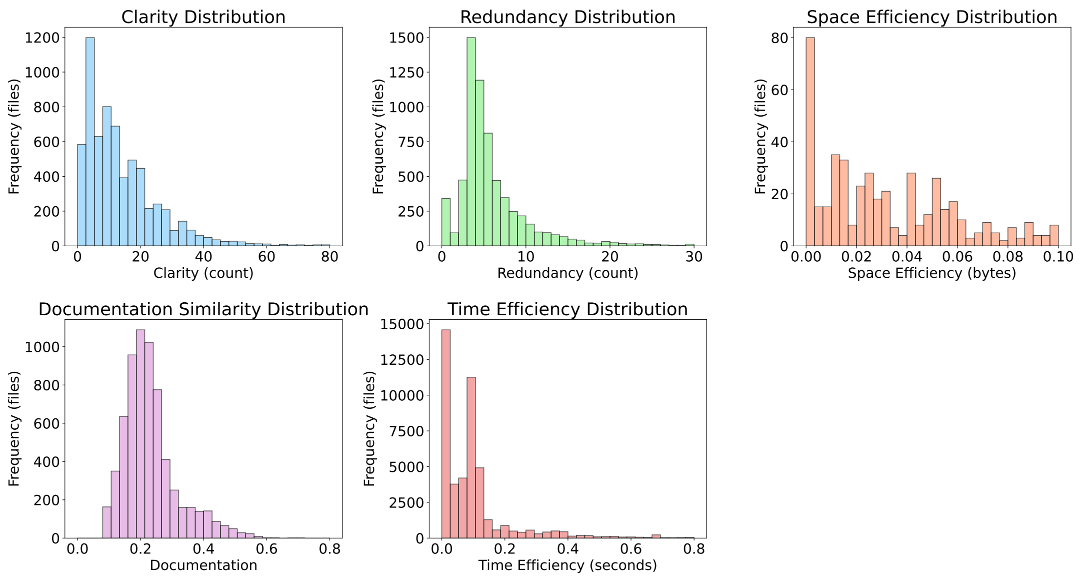


**Figure 1: Distributions of key metrics: clarity, redundancy, time efficiency, space efficiency, and documentation in PIE dataset.**

To ensure data quality, we filtered submissions to include only C++ programs that passed all test cases and met runtime constraints. Each program was paired with its performance-improved version, enabling comparative analysis. We used the Gem5 simulator to normalize execution times, eliminating inconsistencies in raw runtime data. The resulting dataset contains 77,967 pairs of functionally correct programs across diverse problem domains.

## 4.2 Data Analysis and Results

*4.2.1 High-Quality Code Characteristics.* We began the analysis by describing the general characteristics of these C++ codes and deciding what parameters needed to be used in a five-dimensional framework. Programs in the PIE dataset demonstrate strong performance across all five optimization dimensions (Figure 1). Most programs exhibit high clarity (fewer than 10 issues per file) and low redundancy (fewer than 5 redundant constructs), indicating well-structured and efficient code. Space and time efficiency metrics cluster around optimal values, while documentation scores fall within moderate ranges (0.1–0.3).

*4.2.2 Multi-Dimensional Code Quality Analysis.* Table 1 presents our analysis of five key metrics across the PIE dataset. The results reveal distinct patterns:

**Clarity:** 58.65% of files contain fewer than 10 clarity-related issues, indicating generally well-structured code.

**Redundancy:** 32.36% of files show moderate to high levels of redundant constructs, suggesting optimization opportunities.

**Space Efficiency:** 19.19% of files consume significantly more memory than average, highlighting areas for memory optimization.

**Documentation:** 6.30% of files demonstrate high semantic alignment between code and documentation, while 5.60% show minimal documentation efforts.

**Time Efficiency:** 1.41% of files achieve exceptional performance, representing solutions that excel across all metrics.

*4.2.3 Evaluation Strategies Align with the evaluation framework.* Inspired by recent prompting strategies [62] and offline reinforcement learning(offline-rl) techniques [12], we introduced performance tags to evaluate C++ programs. This tagging scheme enables us to track how programmers adjust their optimization strategies over time. For time efficiency, we assigned tags by associating each "fast" program with an indicator of its optimal achievable performance across all solutions in the dataset. Specifically, the tag represents how close a program is to peak performance on a binned scale from 1 to 10, with the top 10% of optimized solutions for a given task labelled as "10/10," the next 10% as "9/10," and so on. Similarly, for space efficiency, tags are assigned based on memory usage during execution, with the top 10% of most memory-efficient solutions labeled "10/10," the next 10% as "9/10," and so on. This reflects how efficiently a solution uses resources.

As shown in Table 1, for other metrics (clarity, redundancy, and documentation), which lack direct quantitative parameters, five members of the research team analyzed the code files to identify specific evaluation parameters within each category. For clarity, we assess magic literals, naming consistency, and statement length based on static analysis. For documentation, the distribution was calculated embedding- based semantic similarity between code and documentation. Redundancy detection identifies duplicate code constructs and unnecessary operations(e.g., code constructs redundancy, dead code, and unused code). These thresholds follow established methods in code quality evaluation [14, 47], ensuring our scoring framework aligns with industry standards. Each factor is converted to its percentile rank in the training set, with lower-is-better measures (runtime, memory, redundant lines) inverted (see Table 1 for formulas and before/after examples). These percentiles are combined with predefined weights and then rounded up to a 1-to-10 scale by computing $\lceil 10 \times perc \rceil$.

The final step applies the same ten-group scale: the top 10% for any metric score 10, the next 10% score 9, and so forth. This

**Table 1: Evaluation methods, formulas, and optimization examples for each dimension. Notation: $perc(\cdot)$ returns a training-set percentile rank in $[0, 1]$; $T_{exec}$ is execution time; $M_{usage}$ is peak memory usage; $Redundant_{lines}$ counts duplicate/dead lines. For clarity, $f_i$ are style factors with weights $w_i$ ($\sum_i w_i = 1$). For documentation, $S_{sim}$ is the cosine similarity between a code segment and its accompanying natural-language documentation (e.g., CodeBERT embeddings). The ceiling operator $\lceil \cdot \rceil$ maps the $0-1$ value onto the common $1-10$ rubric (top 10% receive 10).**

| Metric | Formula | Evaluation Method | Examples (Original → Optimized) |
|---|---|---|---|
| **Time Efficiency** | $Score = \lceil 10 \cdot (1 - perc(T_{exec})) \rceil$ | Algorithmic complexity and runtime behavior | `pow(x, 2)` → `x * x` |
| **Space Efficiency** | $Score = \lceil 10 \cdot (1 - perc(M_{usage})) \rceil$ | Memory footprint and data structure usage | `vector<vector<int>> mat(n, vector<int>(n, 0))` → `vector<int> mat(n * n, 0)` |
| **Clarity** | $Score = \left\lceil 10 \cdot \left(1 - \sum_i w_i\, perc(f_i)\right)\right\rceil$ | Magic literals, naming consistency, and statement length<br>Descriptive names and magic number avoidance | `int a = 0;` → `int sum = 0;`<br>`int f(int x);` → `int computeSquare(int x);` |
| **Documentation** | $Score = \lceil 10 \cdot perc(S_{sim}) \rceil$ | Embedding-based semantic similarity between code and documentation | `// calculate result` → `// Computes total revenue from all entries` |
| **Redundancy** | $Score = \lceil 10 \cdot (1 - perc(Redundant_{lines})) \rceil$ | Duplicate code and dead code detection | `for (...) print(x);`<br>`for (...) print(x);`<br>→ `for (...) print(x);` |

common rubric keeps all five metrics on a comparable 1–10 scale and aligns them with industry standards.

## 5 MACEDON: Design and Implementation

The results of our formative study and data analysis confirmed the need for multi-dimensional code evaluation and optimization. This led us to design and develop MACEDON, addressing the first part of **RQ2**, which aimed at assisting novice programmers in improving their code efficiency, readability, and maintainability via the multi-dimensional framework.

### 5.1 System Architecture

The MACEDON system consists of two main components: 1) a client-side UI, implemented as a Visual Studio Code (VS Code) extension using TypeScript, and 2) a server-side backend, developed in Python and Flask.

The client-side program is responsible for rendering the UI and tracking user interactions with the code editor. The client-side component is responsible for rendering the optimization dashboard and tracking user interactions within the code editor. When a user modifies a code segment, MACEDON sends the updated content to the server-side program via HTTP requests for real-time evaluation.

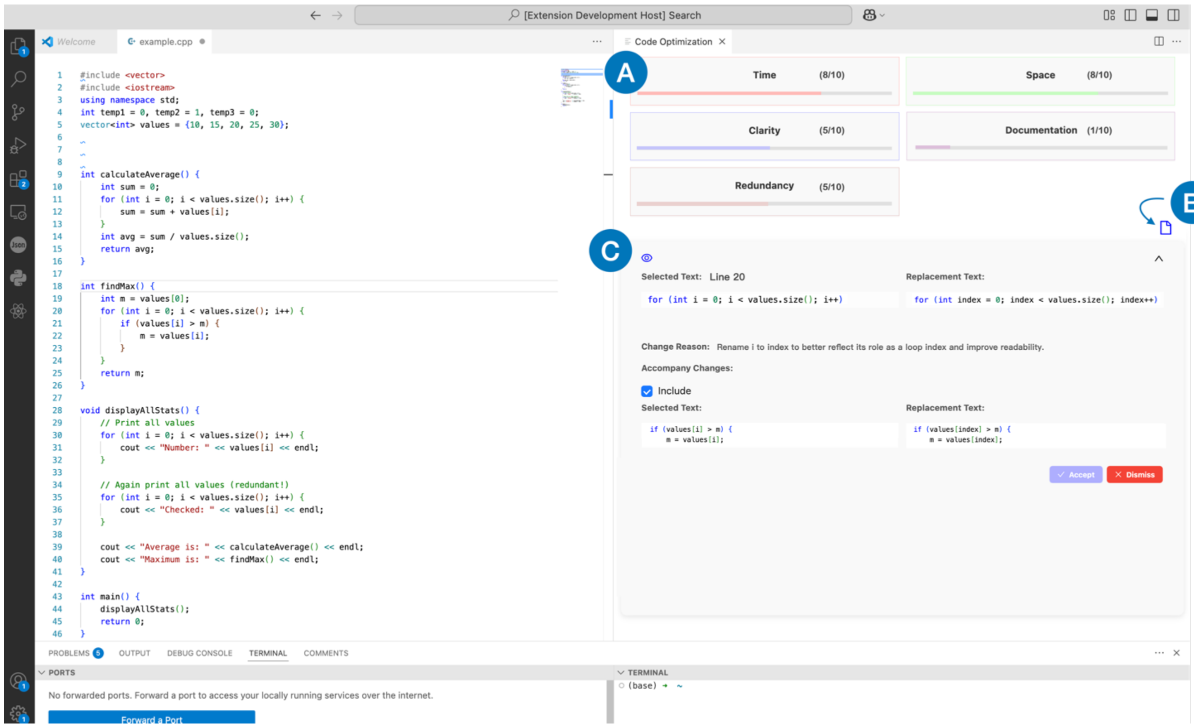
The server-side processes the code using a multi-dimensional analysis framework that evaluates performance, readability, redundancy, and documentation quality. For performance optimization, the server utilizes benchmark-based profiling and heuristic-guided transformations to suggest improvements. For readability and redundancy detection, MACEDON employs a static analysis pipeline to identify redundant constructs and enhance clarity. Additionally, for documentation generation, MACEDON integrates natural language processing (NLP) techniques, extracting context from the code and generating inline explanations. The system also supports cascading updates, ensuring that optimization changes propagate across dependent code segments for consistency.

### 5.2 User Interface Design

Figure 2 shows the UI of MACEDON as a VS Code Extension, which comprises two main UI components.

The Evaluation Dashboard presents a multi-dimensional analysis of the code in real time, covering key dimensions including time efficiency, space efficiency, clarity, redundancy, and documentation (Figure 2.A). Each dimension is visualized with a unique color (red for time efficiency, purple for documentation, blue for clarity, green for space efficiency, and pink for redundancy), enabling users to quickly identify which aspects of the code require attention.

The Optimization Dashboard provides a prioritized list of actionable suggestions based on the real-time evaluation results (Figure 2.C). Suggestions are grouped by dimension and ranked by impact, offering users a clear entry point for making improvements. When a user clicks on a suggestion, an optimization panel appears, showing the original code, the proposed change, and the rationale behind it (Figure 2.C). Users can accept changes with a single click or further customize them. For example, they may rename a variable to enhance clarity or replace a nested loop for improved time efficiency.



```
1   #include <vector>
2   #include <iostream>
3   using namespace std;
4   int temp1 = 0, temp2 = 1, temp3 = 0;
5   vector<int> values = {10, 15, 20, 25, 30};
6
7
8
9   int calculateAverage() {
10      int sum = 0;
11      for (int i = 0; i < values.size(); i++) {
12          sum = sum + values[i];
13      }
14      int avg = sum / values.size();
15      return avg;
16  }
17
18  int findMax() {
19      int m = values[0];
20      for (int i = 0; i < values.size(); i++) {
21          if (values[i] > m) {
22              m = values[i];
23          }
24      }
25      return m;
26  }
27
28  void displayAllStats() {
29      // Print all values
30      for (int i = 0; i < values.size(); i++) {
31          cout << "Number: " << values[i] << endl;
32      }
33
34      // Again print all values (redundant!)
35      for (int i = 0; i < values.size(); i++) {
36          cout << "Checked: " << values[i] << endl;
37      }
38
39      cout << "Average is: " << calculateAverage() << endl;
40      cout << "Maximum is: " << findMax() << endl;
41  }
42
43  int main() {
44      displayAllStats();
45      return 0;
46  }
```

No forwarded ports. Forward a port to access your locally running services over the internet.

```
(base) → ~
```

Time (8/10)
Space (8/10)
Clarity (5/10)
Documentation (1/10)
Redundancy (5/10)

Selected Text: Line 20

```
for (int i = 0; i < values.size(); i++)
```

Replacement Text:

```
for (int index = 0; index < values.size(); index++)
```

Change Reason: Rename i to index to better reflect its role as a loop index and improve readability.

Accompany Changes:

[✓ Include]

Selected Text:

```
if (values[i] > m) {
    m = values[i];
```

Replacement Text:

```
if (values[index] > m) {
    m = values[index];
```

**Figure 2: MACEDON is an extension of the Visual Studio Code. It provides three basic dashboards: (A) Evaluation Dashboard provides the current score of their code in five dimensions(clarity, documentation, redundancy, time efficiency, and space efficiency); (B) a button for user to trigger Optimization Dashboard; (C) Optimization Dashboard provides a prioritized list of actionable suggestions based on the real-time evaluation results in each dimension.**

## 5.3 Approach for Multi-Dimensional Code Evaluation and Optimization

*5.3.1 Code Evaluation.* Our formative study suggests that the system should support structured, multi-dimensional code evaluation, covering key aspects such as performance, readability, redundancy, space efficiency, and documentation quality to provide programmers with a comprehensive understanding of their code.

Some types of code evaluation can be directly derived from static analysis, allowing for automated assessments. For example, clarity can be analyzed using Clang-based static analysis, identifying areas where variable naming, indentation, or code structure could be improved for better readability. Similarly, time efficiency and space efficiency can be assessed through runtime profiling, where CPU execution time and memory consumption are monitored to detect performance bottlenecks and inefficient memory usage.

Other types, including redundancy and documentation quality, require deep learning-based approaches to capture more abstract patterns. MACEDON leverages embedding-based similarity analysis to detect redundant code structures by comparing functionally similar but structurally different code blocks. For documentation evaluation, MACEDON uses a semantic similarity model to analyze the alignment between existing code comments and the corresponding implementation, identifying areas where documentation is incomplete or inconsistent. Through this hybrid evaluation approach, MACEDON provides targeted optimization recommendations while maintaining a balance between automated insights and user-driven refinements.

**Deep Learning Approach:** We propose a deep learning approach to detect code redundancy and evaluate documentation quality. For redundancy detection, our method leverages a hybrid embedding model that captures both syntax and semantics by processing Abstract Syntax Trees (ASTs) alongside tokenized code. This multi-level analysis enables the detection of identical fragments as well as functionally similar yet structurally diverse code. We trained a Graph2Seq [57] model on 8,500 manually- and automatically-labeled code pairs using contrastive loss over 120 epochs on an H100 GPU cluster. Our approach achieved an F1-score of 87.3%, outperforming traditional token-based (72.5%, e.g., CCFinder [26]) and AST-based (81.2%, e.g., Deckard [24]) methods.

For documentation quality evaluation, a transformer-based model (e.g., CodeBERT) was fine-tuned on 12,000 code-comment pairs, including synthetic paraphrased data. This model assesses whether inline comments and docstrings adequately explain the corresponding code. Evaluated with BLEU, METEOR, and BERTScore metrics, the model achieved a BERTScore of 0.84, demonstrating its ability to capture semantic relevance even when lexical differences exist.

A qualitative analysis on 50 random code snippets indicated that while our model occasionally overestimates redundancy in boilerplate code and under-scores concise but sufficient documentation, it generally distinguishes between necessary repetitions and truly redundant patterns, as well as highlights missing explanations for complex logic.

**Statistic Anaysis Approach:** Our formative study shows that key aspects of code quality—clarity, time complexity, and space complexity—can be effectively evaluated using static analysis without deep learning.

We use Clang-based static analysis to identify readability issues such as inconsistent naming, deeply nested loops, hardcoded values, ambiguous function names, and poor indentation. Clang's AST parsing also detects complex control structures and suggests refactoring strategies.

Time complexity is estimated by profiling loop structures and function calls. By analyzing the AST, our system extracts nested loops, recursion, and dependencies to approximate computational complexity based on known algorithm patterns. A cost-modeling approach further estimates execution time from static operation counts, eliminating the need for runtime profiling.

Space complexity is assessed by examining memory allocations. Our analysis tracks variable sizes, pointer usage, and dynamic allocations through Clang's type analysis, distinguishing between heap and stack usage. Redundant copying and memory leaks are flagged, with recommendations for optimizations such as pass-by-reference or in-place modifications.

This integrated static analysis framework offers a lightweight yet effective method for evaluating code quality and can be extended to other programming languages by adapting AST parsing and memory tracking techniques.

#### 5.3.2 *Code Optimization.*

We fine-tuned a transformer-based model, Qwen-Coder[1], for multi-dimensional code optimization using a targeted dataset rather than large-scale data. Instead of relying on intuitive few-shot in-context learning, we adopt a prefix-tuning supervised approach by clearly stating the desired optimization dimension (e.g., "Clarity:", "TimeEfficiency:") in the prompt.

For data preparation, we curated approximately 500–800 code edits from public GitHub commits. Each commit was filtered using heuristic keywords (e.g., "refactor," "speed up," "optimize memory," "remove duplication," "improve docs") and manually validated to ensure the edits reflected genuine improvements across five dimensions: clarity, time efficiency, space efficiency, redundancy, and documentation.

We fine-tuned a single multi-task model over 60 epochs (batch size 16) with early stopping on a single H100 GPU. The unified model learns dimension-aware code transformations, allowing users to steer the optimization by specifying the target dimension in the prompt during inference. This design supports local, iterative deployment, where developers receive real-time, tailored code improvement suggestions.

Evaluations were conducted using both quantitative and qualitative metrics. For clarity, the Maintainability Index (MI) increased from 68.2 to 81.7, while Cyclomatic Complexity (CC) dropped from 4.6 to 3.1 on a test set of 100 samples. Time efficiency improvements included an average 1.82× speedup, with 63% of cases showing reduced algorithmic complexity (e.g., from $O(n^2)$ to $O(n \log n)$). Memory profiling revealed an average reduction of 27.4% in memory usage. Redundancy metrics showed a 42.5% decrease in redundant code lines, and documentation quality, measured using a fine-tuned CodeBERTScore model, improved to a BERTScore of 0.842, outperforming standard models.

[1]https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct

The MACEDON dashboard integrates these metrics—performance, clarity, and redundancy—allowing programmers to visualize and directly manipulate their code quality assessments.

#### 5.3.3 *Facilitating Code-Optimization Correspondence.*

MACEDON offers features that enable programmers to navigate various code optimization suggestions while maintaining alignment between the code and its corresponding feedback blocks (DG3).

**Showing Corresponding Code.** The code editor highlights relevant code segments based on the selected optimization suggestion. Each optimization dimension, such as performance, readability, or space efficiency, is represented with a distinct color. For example, clarity issues may be highlighted in blue, while space efficiency suggestions appear in purple. This ensures that programmers can quickly identify which part of the code relates to a specific optimization, with other code sections folded to reduce visual clutter.

**Dependency Highlight and Updates.** When code segments are modified, Macedon triggers related updates across dependent code sections, ensuring consistency. These dependent code segments are highlighted in distinct colors based on the type of update, with opacity indicating the relevance of each segment to the original suggestion. For example, renaming a variable will not only update that specific segment but will also highlight related usages throughout the code, with clearer visibility on more relevant instances. This helps the programmer track how a single change impacts the rest of the code, maintaining consistency across the entire codebase.

## 6 User Evaluation

To further investigate **RQ2** regarding MACEDON's effectiveness, we conducted a within-subject controlled experiment which aimed at understanding (1) how well it facilitates code optimization across multiple dimensions and (2) how programmers perceive the multi-dimensional optimization strategy.

### 6.1 Participants

We recruited 24 novice programmers (16 males, 8 females; $MD = 20$, $SD = 3.54$) to participate an in-lab user study evaluating the MACEDON. All participants had less than one year of programming experience and C++ proficiency scores of 2 or lower on a 5-point scale ($MD = 1.42$, $SD = 0.51$). To ensure familiarity with LLM-based code assistants (e.g., GitHub Copilot), we screened participants using a self-assessed 5-point Likert scale measuring familiarity ($MD = 4$, $SD = 0.74$) and required prior experience with code generation using such tools, which participants reported using regularly ($MD = 8$ times/week, $SD = 2.56$). We used a snowball sampling approach to recruit participants, where we sent recruitment messages to friends, colleagues, and various university mailing lists. We then asked participants to refer their friends and colleagues. Following the tests, we conducted semi-structured interviews and distributed questionnaires to gather user feedback.

### 6.2 Study Protocol

A within-subject design was employed. Participants' task was to evaluate and optimize a C++ code given by us. The task focused on evaluating MACEDON's core feature—multi-dimensional code optimization. Without being explicitly provided with guidance on editing, participants were asked to improve their code until all five quality scores (clarity, documentation, redundancy, time efficiency, and space efficiency) reached $\geq 8$. This setup avoids bias and preserves open-ended exploration while ensuring comprehensive evaluation across all dimensions. To simulate a realistic development context, participants were told that they were optimizing the code for the purpose of sharing code for their professor or interviewer who needs to evaluate their code and give them scores. Each participant is asked to finish two sessions, one with the MACEDON support and one with Copilot. We prepared two draft C++ codes, one for each session. The two experimental C++ code are adapted from Github. The two C++ codes have a similar length and a similar level of difficulty. To counterbalance the order effect, we randomized the order of the Copilot condition and the MACEDON condition for each participant, so some participants encountered MACEDON in their first session, and some others experienced it in their second session. This way, we could investigate the capabilities of MACEDON from multiple perspectives more thoroughly.

Each session was limited to 30 minutes to finish one session. We conducted three pilot runs in which all pilot participants were able to complete the tasks within 20 minutes, with MACEDON or with Copilot. Prior to the MACEDON condition session, participants were given a 2-minute quick demo of MACEDON 's interface and features. All study sessions were conducted remotely over a video conferencing tool. We asked participants to share their screens and we video recorded the entire session with their permission. After completing both sessions, we conducted a post-study semi-structured interview to gather qualitative feedback. Participants were asked questions such as "How did MACEDON 's evaluation and optimization compare with the baseline tool in helping you improve code?" and "Which optimization dimensions did you find most useful or challenging?" Additionally, participants were encouraged to share their reflections on system usability, tell their stories, and experience outside these structured questions. The interview sections of the video recordings were transcripted into text.

## 6.3 Data Collection and Measurement

We collected three types of data sources: the screen recordings and observational notes for each session, the final optimized code artifacts from each session, and the post-task questionnaire responses and interview transcripts.

Our first group of measurements focuses on behavioral data extracted from session recordings. For each session, we recorded the task completion time (in seconds). In the condition with MACEDON support, we also counted: how many times a participant triggered MACEDON (e.g., clicked on a specific optimization tag); how many times the participant adopted a suggestion generated by MACEDON (code lines directly modified using system suggestions); how many times the participant ignored the system-generated recommendation and manually edited the code (manual optimizations); and how many times the final version of the C++ code is co-created by users and MACEDON.

Second, to evaluate the quality of the final C++ Code artifact, we define our second group of measurements by counting: the number of lines changed and the total number of tokens changed by MACEDON or Copilot, the total number of tokens changed by humans, as these three are indicators of the quantity and effort each participant spent on the C++ Code. In addition, we asked participants to rate their own satisfaction with the final optimized code for each condition using a 5-point Likert scale (-2 to 2). To objectively assess optimization quality, we asked two C++ experts (C++ Programming Years: $MD = 10.1$ years, $SD = 2.34$) to rate the C++ code-level quality using a 3-dimensional rubric measuring improvements in readability, maintainability, and performance. Each dimension was scored on a scale from -2 to 2. Two experts iteratively discussed and evaluated the C++ code until the independent ratings achieved an agreement (Krippendorff's alpha: $\alpha = 0.76$). The result of this analysis is reported in Table 2.

Finally, we asked the participants to finish a post-experiment survey (5-point Likert Scale, -2 as strongly disagree to 2 as strongly agree) to assess participant perceptions of MACEDON across multiple factors, including usability, accuracy, trust, satisfaction, and adoption propensity based on [55]. These survey results are summarized in Figure 3. For the interview transcripts, three researchers of this project conducted an iterative open coding method to get the code, theme, and representative quotes as another group of data. They each independently coded a subset of interview transcripts and discussed the codes together. Through iterative discussion and re-coding, we applied the codes and themes to their assigned code samples. Some examples of the identified themes include: pros and cons of MACEDON, experiences with specific optimization dimensions, preference of the multi-dimensional code evaluation and optimization approaches, expectations for future adoption, and suggestions for system design improvement. These qualitative insights are reported with quantitative findings to provide a comprehensive view of user experience with MACEDON.

## 6.4 Results

In this section, we present the results of our user study, focusing on how MACEDON improves participants' performance on the task, how participants perceived the multi-dimensional code evaluation and optimization methods in MACEDON, and how participants described the usability and effectiveness of MACEDON.

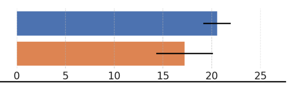
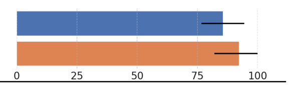
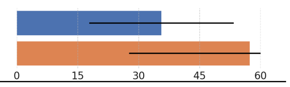
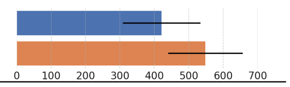
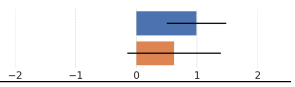
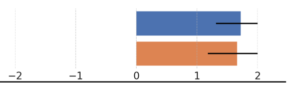
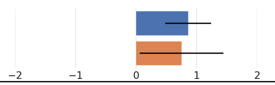
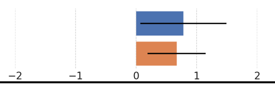
### 6.4.1 *MACEDON Supports Participants to Easily Optimize the Code.*

Our study shows that MACEDON improved participants' code optimization performance by reducing task completion time and enhancing the satisfaction of the final code.

We conducted a two-way repeated measures ANOVA to examine the effect of tool usage (with MACEDON or with Copilot) on task completion time. As shown in Table 2, participants completed the optimization tasks significantly faster with MACEDON ($M = 421.17$, $SD = 112.45$) compared to the condition using Copilot ($M = 548.33$, $SD = 108.61$). Besides, there was not significant effect of codes was found on task completion time.

The post-experiment survey result further supported our findings. Most participants agreed that MACEDON helped them finish the task with MACEDON's help (23 out of 24 rated agree or higher),

**Table 2: Performance data in two conditions (M: mean, SD: standard deviation): the task completion time (secs), participants' satisfaction with the final code artifact (from -2 to 2), code quality, number of line, and number of tokens. In particular, participants spent less time to complete the task in the MACEDON condition than the Copilot condition (p = .001); participants were more satisfied with the final Code in the MACEDON condition than the Copilot condition (p = .001).**

| | Condition | M | SD | |
|---|---|---|---|---|
| Number of Line changes | MACEDON | 20.54 | 1.41 |  |
| | Copilot | 17.21 | 2.90 | |
| Number of token changed by tools | MACEDON | 85.57 | 8.89 |  |
| | Copilot | 92.26 | 10.24 | |
| Number of tokens human changes | MACEDON | 35.62 | 17.81 |  |
| | Copilot | 57.32 | 29.64 | |
| Task Completion Time (secs) | MACEDON | 421.17 | 112.45 |  |
| | Copilot | 548.33 | 108.61 | |
| Satisfaction with the Final C++ Code (-2 to 2) | MACEDON | 0.99 | 0.49 |  |
| | Copilot | 0.62 | 0.77 | |
| Expert Rating: Readability (-2 to 2) | MACEDON | 1.72 | 0.41 |  |
| | Copilot | 1.66 | 0.48 | |
| Expert Rating: Maintainability (-2 to 2) | MACEDON | 0.86 | 0.38 |  |
| | Copilot | 0.75 | 0.69 | |
| Expert Rating: Performance (-2 to 2) | MACEDON | 0.78 | 0.71 |  |
| | Copilot | 0.67 | 0.48 | |

as shown in the Figure 3. Participants also found the code evaluation and were accurate (19 out of 24). Participants noted that the integration into VS Code and the dimension-tagged interface enabled them to optimize specific code segments directly, thus it was easier for them to start optimizing the code: *"It's not just about fixing bugs, it shows me where things can improve — like this loop can be faster."* -P4

#### 6.4.2 *Multi-dimensional Code Evaluation in MACEDON Yields Better Quality of Code and Improves Accuracy and Readability.*

Through coding the video recordings for two sessions, we were able to examine the following questions: While the MACEDON was available, how did the participants use it, especially multi-dimensional code evaluation during code optimization? Did they check the evaluation scores from MACEDON when optimizing the code? Did they actually use those code recommendations under different dimensions when optimizing the code? Did they adopt them directly, or did they revise them for their specific needs?

Based on our findings, when MACEDON was available, most participants (83.3%) viewed evaluation scores at first by clicking on the MACEDON extension in the Visual Studio setting. Most participants (91.7%) not only checked these scores, but also opened the recommendation panel linked to the evaluated dimension generated by MACEDON.

We also found that each participant triggered recommendations ($M$ = 14.2, $SD$ = 4.1), directly adopted suggestions without modification ($M$ = 8.9, $SD$ = 3.6), ignored those deemed unhelpful ($M$ = 3.2, $SD$ = 2.5), and co-created solutions by modifying suggestions before accepting them ($M$ = 2.1, $SD$ = 1.4). Overall, 62.5% of final edits were adopted directly from MACEDON's suggestions, demonstrating strong reliance on system-generated feedback. As P7 described, *"It gave me a starting point, like changing a loop, but then I tweaked the logic to better fit my approach."* An interesting pattern emerged regarding participants' optimization workflows. While individual preferences varied, most participants followed a structured sequence when applying improvements. Specifically, 15 out of 24 participants began with either redundancy or clarity, followed by space and time efficiency, and ended with documentation. The three most common sequences were: (1) redundancy → clarity → space efficiency → time efficiency → documentation (9/24), (2) clarity → redundancy → space efficiency → time efficiency → documentation (6/24), and (3) time efficiency → space efficiency → redundancy → clarity → documentation (6/24). (4) space efficiency → time efficiency → redundancy → clarity → documentation (6/24).

In order to explore the differences between workflows that start with redundancy/clarity versus those that start with space/time efficiency, start from documentation, we conducted a code-level expert rating along the dimension of readability, maintainability, and performance. Each expert needs to rate 48 code samples generated from Copilot and MACEDON. Specifically, readability is the extent of which a human reader can understand the purpose, control flow, and structure of the code. Maintainability describes how

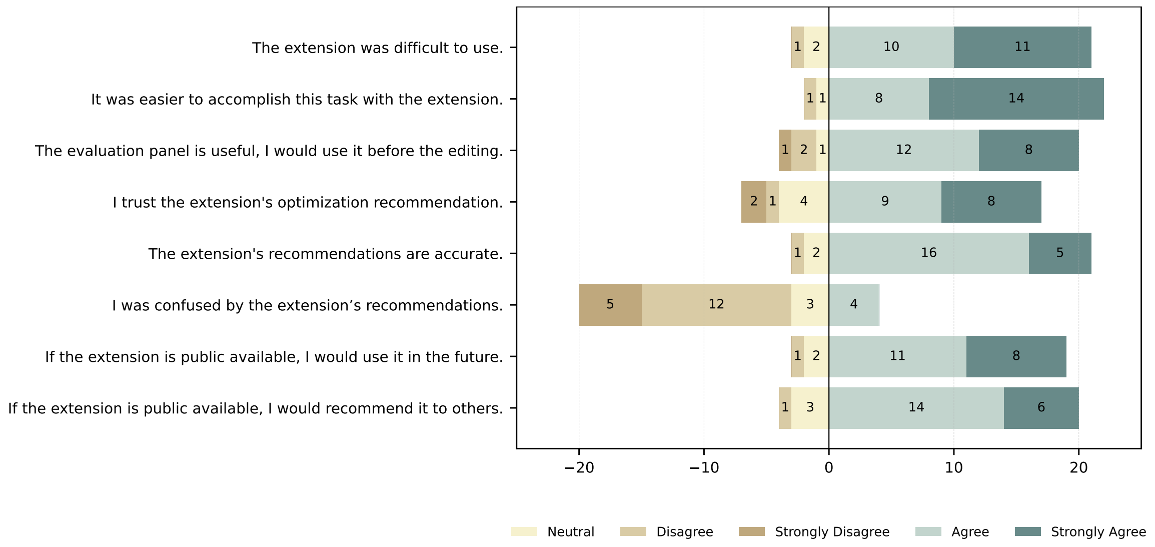


**Figure 3: Results of the post-task questionnaire. The numbers indicate the counts of participants on corresponding scales.**

easy it is to understand, modify, extend, and fix the code over time. Performance describes the code's efficiency in terms of runtime speed and memory usage. We also calculated the token count of the code. We performed a one-way ANOVA to examine the differences among the three groups. As shown in Table 2, participants who optimize the code starting with redundancy or clarity yielded significantly higher ratings in both readability ($F = 8.48, p < 0.001$) and maintainability ($F = 1.25, p < 0.001$), while for performance, there are no significant differences across the three groups. Our posthoc analysis also suggested that no significant differences were found between code optimized starting from space/time efficiency and documentation only along all dimensions (including readability, maintainability, and performance).

*6.4.3 MACEDON increases participants' satisfaction.* The post-task questionnaire revealed that participants were significantly more satisfied with the code optimization experience when using MACEDON condition compared to the condition with Copilot ($F = 3.46, p <$ .001 (Figure 3). P3 mentioned that the dimension-based optimization recommendation helped them feel more in control: *"I have an overview of what to change. This actually gave me a checklist, which made things easier."*

Some participants also highlighted that MACEDON reduced their thinking load during optimization. Instead of needing to decide where to start or what to fix first, MACEDON reminds them to edit the code that they may ignore. *"It's like having a to-do list built into the code. I didn't feel lost."* -P8

Moreover, participants believed that MACEDON can help them form a better optimizing code practice in the long term: *"It is useful to remind me which aspects of the code I usually overlook in a timely manner."* Even though the quality scores for final code artifacts rated by two expert reviewers across four dimensions (clarity, time efficiency, redundancy, documentation) were not significantly different between the MACEDON and Copilot. (Figure 3), participants(P11-14) reported a smoother and more confident experience using MACEDON. As P11 noted, *"I've had a much better experience with MACEDON. Most of the time, its optimization suggestions are accurate. What I really like is that it can update all the related code in one go with just a click. Even though sometimes I didn't fully agree with one recommendation, it still made me think, which I found really helpful."* In some cases, we believe that while MACEDON improves programmers' productivity during code evaluation and optimization, it may not always produce code that aligns perfectly with individual coding styles or domain-specific practices. Thus, programmers may still prefer to revise or refine the naming conventions, formatting, and logic structure to ensure the final code fits their own standards and the broader context of their project.

In summary, while our experiment show that MACEDON improves the perceived usability and satisfaction of the code optimization process. By offering real-time, multi-dimensional code evaluation and optimization, MACEDON also enhances productivity and supports a more reflective and satisfying development workflow.

*6.4.4 The multi-dimensional optimization strategies in MACEDON are suitable for different user goals and expertise levels.* In this section, we examine how participants perceived the different dimensions that MACEDON supports during code optimization. During the post-experiment interviews, we introduced the design behind each dimension and asked participants which dimensions they preferred, and in what contexts they found them most useful.

Participants commonly reported that the clarity and redundancy dimensions were useful entry points, especially when first encountering unfamiliar code. These suggestions helped them clean up structure and naming conventions, which improved overall readability. As P6 noted, *"I always start with clarity or redundancy. If I don't understand or forgot my own code, performance won't matter."*

The time efficiency and space efficiency dimensions were considered as more technical and goal-specific from participants. Several participants (P1-3, P7-9 P17-18) appreciated the performance profiling features, but only used them when optimizing for speed or handling large data. *"I only look at time when I know performance* *matters. Otherwise I just skip it since sometimes I'm familiar with high-level algorithm and I'm not sure whether it is correct."* -P8 Many participants prefer optimizing the code in clarity and redundancy first compared to simply improve the space and time efficiency. P10 mentioned, *"I liked that I could ignore performance stuff at first and just focus on cleanup. Later, I switched to the time efficiency tab to improve it."*

The documentation dimension had different reactions from our post-experiment interview. Some participants(P6, P22-23) said they rarely used it since they preferred to document in their own words. Others felt it was helpful to maintain consistency across their code: *"It's a nice reminder, but sometimes I prefer to rewriting it in my way."* -P2

These findings suggest that different dimensions in MACEDON cater to different goals: clarity and redundancy help with code understanding and onboarding, while performance dimensions are more aligned with production-quality code. In particular, novice programmers benefited from a structure that guided them from readable to efficient code in a step-by-step way.

Participants (P11-13, P20-24) also highlighted that their optimization strategies depended on different scenarios. When writing code for themselves, they focused more on clarity. When sharing or submitting the code, they paid more attention to performance and documentation. As P11 summarized, *"I care if it works and is readable for my own stuff. I care about speed and explaining it better if I need to share with my friends or teachers."*

These insights indicate that future versions of MACEDON could personalize workflows based on user preferences or experience level. For example, a scenario-aware mode could prioritize readability and clarity during individual or exploratory coding, while switching emphasis on performance and documentation when the code is intended for collaboration or submission.

*6.4.5 Summary of the Results.* In summary, our study shows that MACEDON significantly improves task efficiency and user satisfaction during code evaluation and optimization tasks. Participants actively engaged with the system's feedback in multiple dimensions. Expert evaluation data further suggest that MACEDON helped them produce higher-quality code across multiple dimensions. Overall, participants enjoyed using MACEDON and perceived MACEDON as a valuable assistant for both learning and real-world development workflows, particularly for tasks that require balancing performance and maintainability.

## 7 Case Studies

To further examine how MACEDON generalizes to real-world usage scenarios (**RQ3**), we conducted an in-depth review of two programming cases by inviting two participants to use the system based on their real-life needs in two one-hour hands-on sessions. These two cases span diverse settings, including early-stage programming task and code interview programming. We intend to: (1) illustrate how MACEDON adapts to different user needs and workflows and (2) contextualize the MACEDON framework through practical examples.

### 7.1 Early-Stage Programming Task

To learn how MACEDON can be used in early-stage learning, we recruited a first-year undergraduate student (S1) who had completed only an introductory programming course. As part of her daily assignment for the programming course, she needed to ensure her code had great clarity and great time efficiency when submitting to the teacher, as she wanted to get a good score for this class. The example is shown in Figure 4.

**Exploring Optimization Dimensions.** S1 launched MACEDON from the Visual Studio Code extension interface and began reviewing her assignment code, a sorting algorithm that she had written in a previous lab. She was immediately drawn to the Optimization Dashboard, which highlighted areas of concern using color-coded scores across clarity, redundancy, time efficiency, space efficiency, and documentation.

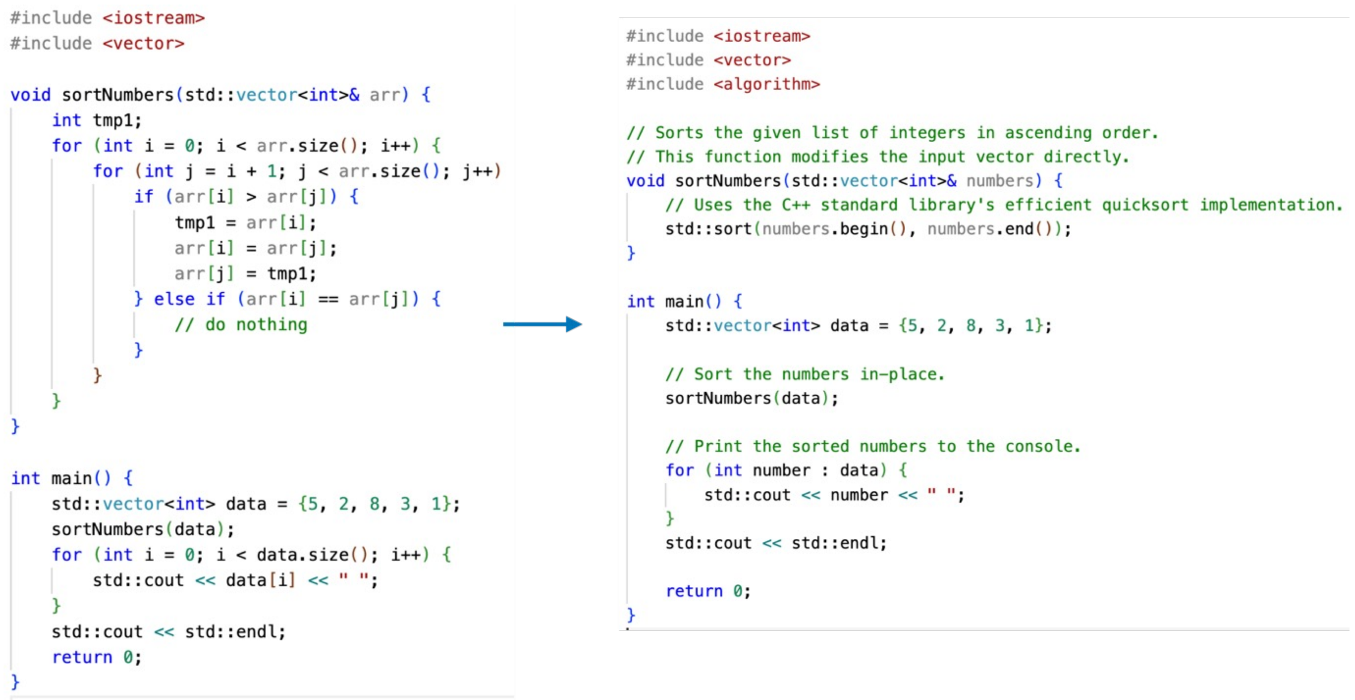
S1 quickly noticed the code had low scores in clarity and redundancy. She clicked into these sections and received immediate feedback, including suggestions to rename variable names like `tmp1` and remove duplicated branches in conditionals. She commented: *"I didn't think 'tmp1' was a problem, but now that I see it flagged, I agree it looks messy."* She then accepted the renaming suggestion and applied a small logic refactor suggested by the tool.

She also explored the time efficiency section, which recommended replacing a nested loop with a more optimized standard library call. S1 was unfamiliar with the suggested function and used the built-in explanation panel to understand its purpose. *"I've never used that function before, but the explanation helped. I can tell this will run faster for big inputs."*

**Customizing Optimization Suggestions and Learning Via Edits.** Once S1 became comfortable with the interface, he began interacting more deeply with the suggestions offered by MACEDON. For each dimension, the system displayed not only suggested edits but also brief rationales and potential trade-offs. In the time efficiency category, MACEDON flagged a linear search loop and suggested using a set for faster lookup. S1 hesitated, and mentioned: *"I've never used a set in C++... won't that make it more complicated?"* The explanation tooltip clarified that using a set would improve performance for large inputs, and provided a short code example.

Motivated by this explanation, S1 attempted to rewrite the loop using the *std::unordered_set* structure. Although he initially struggled with the syntax, he used the integrated documentation feature within MACEDON to reference an example. After some trial and error, he got the revised code to compile and work correctly. *"This is the first time I used a set. I probably wouldn't have tried it if the tool didn't suggest it."* This moment was significant—it illustrated how MACEDON not only suggested improvements but also acted as a learning scaffold, empowering the student to try out unfamiliar constructs.

S1 also customized a few suggestions that he felt were too generic. For example, when MACEDON proposed replacing a loop with a built-in function, P1 opted to preserve the loop but renamed variables and added comments for clarity. This show emerging confidence in balancing tool guidance with personal judgment. The ability to selectively adopt, adapt, or reject suggestions made MACEDON feel less like a rigid evaluator and more like a collaborative assistant.



```cpp
#include <iostream>
#include <vector>

void sortNumbers(std::vector<int>& arr) {
    int tmp1;
    for (int i = 0; i < arr.size(); i++) {
        for (int j = i + 1; j < arr.size(); j++)
            if (arr[i] > arr[j]) {
                tmp1 = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp1;
            } else if (arr[i] == arr[j]) {
                // do nothing
            }
        }
    }
}

int main() {
    std::vector<int> data = {5, 2, 8, 3, 1};
    sortNumbers(data);
    for (int i = 0; i < data.size(); i++) {
        std::cout << data[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

// Sorts the given list of integers in ascending order.
// This function modifies the input vector directly.
void sortNumbers(std::vector<int>& numbers) {
    // Uses the C++ standard library's efficient quicksort implementation.
    std::sort(numbers.begin(), numbers.end());
}

int main() {
    std::vector<int> data = {5, 2, 8, 3, 1};

    // Sort the numbers in-place.
    sortNumbers(data);

    // Print the sorted numbers to the console.
    for (int number : data) {
        std::cout << number << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

**Figure 4: Optimized version of a student's sorting algorithm with improvements guided by MACEDON. Detailed optimization by dimension: (1) Clarity: Variable names such as `arr` and `tmp1` were replaced with more descriptive names like `numbers` and `number`, improving readability. The loop to print sorted values was changed to a range-based for loop for more concise and clearer iteration. (2) Time Efficiency: The original nested loop sorting algorithm with time complexity O($n^2$) was replaced by the C++ standard library function `std::sort`, which uses an optimized quicksort algorithm, improving the sorting time to O(n** log **n). (3) Space Efficiency: Unnecessary temporary variables (e.g., `tmp1`) and redundant conditional checks (e.g., `arr[i] == arr[j]`) were removed. The space overhead was reduced as no extra memory was used for the sorting process. (4) Documentation: Added comments explaining the function's purpose, the choice of sorting algorithm, and the logic of the iteration and output steps. These comments make the code easier to understand for beginners and future developers. (5) Redundancy: A redundant conditional branch `else if (arr[i] == arr[j])` was removed because it had no functional effect, simplifying the logic and reducing unnecessary operations.**

**Reflecting on Learning and Building Confidence.** At the end of the session, we asked P1 to reflect on his overall experience with MACEDON. He emphasized that the most helpful part was seeing specific suggestions categorized by dimension: *"It's different from getting a bunch of corrections at once. I can focus on one thing at a time—first make it cleaner, then think about speed."* This multi-dimensional recommendation helped him develop a more systematic approach to refining code, which was previously absent from his workflow.

S1 also appreciated that MACEDON gave him the freedom to make the final call. He said: *"It's not like the tool is telling me I'm wrong. It's more like: here's something you might not have thought about."* He expressed a willingness to use MACEDON in future assignments, particularly for reviewing his code before submission. He also noted that seeing his own improvements logged in the history panel boosted his confidence: *"It's cool to see all the changes I made. Feels like I'm actually getting better."*

Finally, S1 mentioned that he would recommend MACEDON to his classmates, especially those who struggle with identifying what to improve in their code. He concluded: *"We're all still figuring things out. Having something like this—it doesn't just fix your code, it shows you how to think about it differently."* The case study demonstrates how MACEDON can serve not only as a productivity enhancer, but also as an educational ally, shaping how novice programmers learn to reason about quality in code.

## 7.2 Programming Evaluation During the Interview

To assess MACEDON's utility for experienced developers, we invited a senior software engineer (S2) who had 10+ years of experience in backend systems and frequently conducted technical interviews. As part of her work, she had to evaluate the C++ code written by interview candidates.

**Rapid Multi-Dimensional Evaluation.** S2 began by pasting a recent candidate's C++ submission into the Visual Studio and launched MACEDON to evaluate the code's quality. He immediately noted the dimension-based feedback provided by the Optimization Dashboard. Color-coded bars surfaced low-performing dimensions, particularly redundancy and clarity. He appreciated that each suggestion was paired with a concise explanation and visual highlights in the editor: *"This kind of feedback is exactly what I usually provide during interviews, but it's great to see it automatically structured like this."*

He found the breakdown by dimension particularly useful in identifying overlooked issues. For example, MACEDON pointed out a duplicated loop construct and variable names like `res1` and `temp2` that reduced readability. He noted that *"These are things I'd catch manually, but having the tool flag them saves time. It's like a second pair of eyes."* He also mentioned that the time and space efficiency scores helped him quickly understand whether the solution was over-engineered or underperforming—without running the code. *"Just looking at complexity isn't enough. These metrics back that up with context."*

**Comparing Candidate Submissions.** Next, S2 opened a second candidate solution for the same problem to compare the two implementations side-by-side. He find the evaluation panel is very intuitive for him to directly know the performance of the current code in four dimensions. He can easily take the screenshot to compare the current candidate's submission to the previous submission by other candidates. He mentioned: *"In interviews, I'm often reviewing 10+ candidates for the same problem. Being able to rank and compare by dimension is incredibly useful."*

S2 used the Optimization Panel to review the candidate's code weakness, which could help him streamline hiring decisions, especially in situations where multiple reviewers assess the same submission. *"This gives me a structured way to explain why one solution is stronger than another. It adds consistency to the evaluation process."*

He also experimented with editing the weaker submission directly in MACEDON using its built-in suggestions. He found that small improvements in naming and redundant logic removal quickly raised the clarity score. *"It's a good teaching tool too. I could use this during live interviews to show candidates how to improve their code."*

**Mentorship and Continuous Use.** Finally, S2 reflected on how MACEDON could support not only hiring but also mentorship. He envisioned using it during onboarding to help junior engineers understand code quality standards within the team. *"I could see using this in code reviews—not as a replacement, but to catch common issues and free me up to focus on architecture and logic."*

He appreciated that MACEDON did not enforce a rigid style but provided adaptive, dimension-based guidance. *"It doesn't tell you 'this is wrong,' it says 'here's what could be clearer or faster,' which fits well with how I coach junior devs."* When asked whether he would use MACEDON in the future, S2 responded affirmatively, especially for repetitive review tasks and mentoring sessions.

In summary, the case study shows that MACEDON offers tangible value for experienced engineers engaged in code review and technical interviews. Its structured evaluation, real-time guidance, and comparison tools can streamline both hiring decisions and educational feedback, supporting quality and efficiency across professional workflows.

## 8 Discussion

### 8.1 Automated Code Review Practices in Software Engineering

While automated code review systems in software engineering largely operate under the assumption of well-structured, production-ready codebases, our experiment with MACEDON reveals that professional programmers operate across a more diverse and context-sensitive spectrum of goals. Through our session with an expert software engineer, we observed that code review in real-world settings often requires balancing trade-offs between clarity, maintainability, and performance, none of which can be linearly optimized. For instance, as our case study and lab experiment show, participants frequently prioritized clarity and redundancy first before addressing performance dimensions, especially when reviewing code authored by junior developers. This mirrors findings in prior studies that emphasize the human judgment required in triaging automated suggestions [45] and customizing review based on code ownership and complexity [7].

In contrast to many existing tools that provide a flat list of suggestions or flag violations against static rulesets [10, 33], MACEDON's dimension-based feedback model encourages structured decision-making across clarity, redundancy, time efficiency, space efficiency, and documentation. Our quantitative analysis demonstrates that when users began optimization workflows with structural dimensions like clarity or redundancy, their final code artifacts achieved significantly higher accuracy and readability ratings, aligning with goals typically emphasized during onboarding and mentoring. This diverges from code review workflows that are primarily focused on defect detection and performance bugs [33, 51].

Furthermore, our observations resonate with recent discussions in the HCI and software engineering communities that advocate for more context-aware, human-centered design in AI-powered developer tools [38]. For example, research on mixed-initiative review systems suggests that trust and user control are critical to adoption [29]. MACEDON supports this by allowing participants to selectively apply, edit, or ignore system suggestions based on their understanding and review goals. This interaction pattern, especially prevalent among expert users in our study, highlights the value of adaptable, goal-driven review interfaces that go beyond one-size-fits-all automation.

Overall, these findings suggest that while AI-assisted review tools have advanced the automation frontier, their practical effectiveness hinges on aligning with developer intentions, review contexts, and team conventions. MACEDON contributes to this ongoing effort by offering a dimensionally scaffolded review process that reflects how real developers reason about and communicate code quality.

### 8.2 Code Optimization in Programming Differs from Traditional Software Refactoring

Code optimization in programming tasks, especially in practical workflows, differs significantly from traditional software refactoring practices. While traditional refactoring typically emphasizes systematic transformation of code structure, including improving modularity, reducing duplication, or renaming for better readability [18], we found that optimization in real-world scenarios is often goal-driven, non-linear, and context-specific.

In our study, participants did not treat optimization as a purely structural clean-up task. Instead, they strategically selected which aspects to prioritize based on their goals. For instance, our experimental data and expert case study revealed a recurring workflow pattern where participants first optimized for clarity and redundancy before focusing on time and space efficiency. This sequential approach highlights a key distinction from refactoring guidelines that typically apply rule-based transformations uniformly [34]. One participant noted: *"If the code isn't understandable, then optimizing for speed won't help. Clarity comes first."* Such remarks reinforce that optimization is not merely mechanical improvement, but an iterative and interpretative process.

Moreover, optimization decisions were often informed by the intended audience or use case. When participants optimized code for instructional or peer-review scenarios, they focused heavily on clarity and documentation. In contrast, when optimizing for deployment or performance benchmarks, participants shifted attention to low-level algorithmic improvements. This context-aware behavior is rarely supported by traditional refactoring tools, which assume a fixed notion of "better" code, often guided by style guides or design patterns [3].

MACEDON was designed to support this flexible optimization process by providing dimension-specific feedback across five key aspects: clarity, redundancy, time efficiency, space efficiency, and documentation. Rather than enforcing uniform refactoring templates, the system allowed users to select dimensions that aligned with their intent. Our evaluation showed that workflows beginning with structural dimensions (clarity/redundancy) produced code with higher accuracy and readability scores, echoing similar calls for context-sensitive tooling in modern IDEs [8].

These findings suggest that programming optimization workflows are not simply micro-refactorings; they are inherently interpretive, shaped by the programmer's goals, audience, and timing. This also has implications for tool design: systems should move beyond static rules and instead scaffold decision-making with contextual and composable suggestions, much like MACEDON does through its multi-dimensional design.

Furthermore, our observations speak to broader discussions in software engineering and HCI about adaptive support for complex workflows [9, 28]. As programming becomes increasingly collaborative and domain-specific, future tooling should reflect the diversity of optimization goals rather than converge on one-size-fits-all models. Supporting exploration, decision tracking, and even divergent optimization strategies may ultimately prove more useful than enforcing consistency alone.

Beyond these workflow-focused observations, we observed a mismatch between objective performance numbers and participants' feelings of improvement. Experts found many optimized versions produced with MACEDON only slightly faster or lighter in memory, yet participants still felt they had made meaningful progress. We think this mismatch is partly due to MACEDON's real-time scoring interface, which shows separate scores for each dimension after every edit. Even small score increases gave users quick feedback, reinforcing their sense of control. Such timely cues can amplify the perceived value of minor gains and should be integral to future optimization tool design. In summary, our study reveals that code optimization in real-world programming differs substantially from traditional software refactoring in both structure and intent. Tools like MACEDON, which account for the multiplicity of goals and adapt to the workflow at hand, are better suited to support modern programming practices.

## 8.3 Limitations and Future Work

Our study has several limitations. First, we focused on novice programmers and short C++ snippets, which may limit generalizability to professional developers or real-world software repositories. Future work should investigate MACEDON's effectiveness on professional-scale codebases and in team-based development settings.

Second, while we evaluated five optimization dimensions, our system currently performs single-objective optimizations. Exploring multi-objective tradeoffs and dynamic prioritization remains an important next step. In real-world scenarios, programmers often need to balance competing goals such as improving performance without sacrificing clarity. Future work could incorporate interactive mechanisms that allow users to define or adjust their optimization priorities based on context or task-specific constraints.

Additionally, while our system surfaces explainable suggestions through structured UI, we did not explore deeper explainability techniques. Future research could investigate how to visualize reasoning behind recommendations or simulate "what-if" scenarios (e.g., "what happens if I accept this suggestion?") This would increase trust and educational value, particularly in classroom or mentoring settings

Finally, although our study revealed interaction-level behaviors, we did not assess long-term learning effects. Future longitudinal studies could examine whether MACEDON helps users internalize optimization principles or risks inducing over-reliance. These directions are critical for advancing MACEDON's educational and practical impact.

## 9 Conclusion

We presented MACEDON, a Visual Studio Code extension designed to support professional programmers in code optimization tasks through real-time, multi-dimensional feedback. The system serves as both a practical tool and a research probe for understanding how developers make trade-offs across different dimensions of code quality including clarity, redundancy, time efficiency, space efficiency, and documentation. The design of MACEDON was guided by a formative study involving six programmers with varying levels of experience, along with a data-driven analysis on a C++ code benchmark. Our controlled user study and two case studies show that MACEDON significantly reduced task completion time while maintaining or enhancing the accuracy, readability, and satisfaction of the final code.

## Acknowledgments

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant

#RGPIN-2020-03966 and the Canada Foundation for Innovation (CFI) John R. Evans Leaders Fund (JELF) #42371.

## References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023). doi:10.48550/arXiv.2303.08774

[2] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. In *Concurrency and Computation: Practice and Experience*, Vol. 22. Wiley Online Library, 685–701.

[3] Vahid Alizadeh, Marouane Kessentini, Mohamed Wiem Mkaouer, Mel Ó Cinnéide, Ali Ouni, and Yuanfang Cai. 2018. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering* 46, 9 (2018), 932–961.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021). doi:10.48550/arXiv.2108.07732

[5] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 675–698.

[6] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[7] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2013. The influence of non-technical factors on code review. In *2013 20th working conference on reverse engineering (WCRE)*. IEEE, 122–131.

[8] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2017. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering* 45, 3 (2017), 261–284.

[9] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1589–1598.

[10] G Ann Campbell and Patroklos P Papapetrou. 2013. *SonarQube in action*. Manning Publications Co.

[11] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3675–3691.

[12] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. 2021. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems* 34 (2021), 15084–15097.

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021). doi:10.48550/arXiv.2107.03374

[14] Yung-Yu Chen, Yu Su, Hsin-Hsiang Chen, Yu-Ting Chung, Tzu-Fu Chen, Jian-Jia Chen, and Tei-Wei Kuo. 2018. Speedoo: Prioritizing performance optimization opportunities. In *Proceedings of the 40th International Conference on Software Engineering*. 60–70.

[15] Shukai Duan, Nikos Kanakaris, Xiongye Xiao, Heng Ping, Chenyu Zhou, Nesreen K Ahmed, Guixiang Ma, Mihai Capota, Theodore L Willke, Shahin Nazarian, et al. 2023. Leveraging Reinforcement Learning and Large Language Models for Code Optimization. *arXiv preprint arXiv:2312.05657* (2023). doi:10.48550/arXiv.2312.05657

[16] Ilker Etikan, Sulaiman Abubakar Musa, Rukayya Sunusi Alkassim, et al. 2016. Comparison of convenience sampling and purposive sampling. *American journal of theoretical and applied statistics* 5, 1 (2016), 1–4.

[17] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A Becker. 2023. My ai wants to know if this will be on the exam: Testing openai's codex on cs2 programming exercises. In *Proceedings of the 25th Australasian Computing Education Conference*. 97–104.

[18] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[19] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=hQwb-lbM6EL

[20] Nat Friedman. 2021. Introducing GitHub Copilot: Your AI Pair Programmer. 2021.

[21] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael R. Lyu. 2025. Search-Based LLMs for Code Optimization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 578–590. doi:10.1109/ICSE55347.2025.00021

[22] JetBrains. 2001. IntelliJ IDEA: The Java IDE for Professional Developers. https://www.jetbrains.com/ Accessed: 2024-10-10.

[23] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.

[24] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE 2007)*. IEEE, 96–105. doi:10.1109/ICSE.2007.30

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

[26] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670. doi:10.1109/TSE.2002.1019480

[27] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2007. Automatic detection of performance regressions. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 103–113.

[28] Amy J Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 344–353.

[29] Matthias Kraus, Nicolas Wagner, and Wolfgang Minker. 2021. Modelling and predicting trust for developing proactive dialogue strategies in mixed-initiative interaction. In *Proceedings of the 2021 International Conference on Multimodal Interaction*. 131–140.

[30] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[31] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2024. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[32] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[33] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21 (2016), 2146–2189.

[34] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.

[35] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–16.

[36] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2332–2354.

[37] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2006. Refactoring: How do software engineers use it?. In *2006 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 511–514.

[38] Brad A Myers, Amy J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52.

[39] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5.

[40] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=iaYcJKpY2B_

[41] William F Opdyke. 1992. Refactoring object-oriented frameworks. *University of Illinois at Urbana-Champaign* (1992).

[42] Russell A Poldrack, Thomas Lu, and Gašper Beguš. 2023. AI-assisted coding: Experiments with GPT-4. *arXiv preprint arXiv:2304.13187* (2023). doi:10.48550/arXiv.2304.13187

[43] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter F Tichy. 2000. Quantitative assessment of the effectiveness of software development techniques. *Empirical Software Engineering* 5, 3 (2000), 219–249.

[44] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2).* https://openreview.net/forum?id=6vZVBkCDrHT

[45] Maithra Raghu, Katy Blumer, Greg Corrado, Jon Kleinberg, Ziad Obermeyer, and Sendhil Mullainathan. 2019. The algorithmic automation problem: Prediction, triage, and human effort. *arXiv preprint arXiv:1903.12220* (2019). doi:10.48550/arXiv.1903.12220

[46] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 491–514.

[47] Simone Scalabrino, Gabriele Bavota, Barbara Russo, Rocco Oliveto Penta, and Andrea Marcus. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process* 30, 12 (2018), e1958.

[48] Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2024. Learning Performance-Improving Code Edits. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=ix7rLVHXyY

[49] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Prem Devanbu, and Toufique Ahmed. 2025. Calibration and Correctness of Language Models for Code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 540–552. doi:10.1109/ICSE55347.2025.00040

[50] Gareth Terry, Nikki Hayfield, Victoria Clarke, Virginia Braun, et al. 2017. Thematic analysis. *The SAGE handbook of qualitative research in psychology* 2, 17-37 (2017), 25.

[51] Adam Tornhill and Markus Borg. 2022. Code red: the business impact of code quality-a quantitative study of 39 proprietary production codebases. In *Proceedings of the International Conference on Technical Debt*. 11–20.

[52] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2015. JDeodorant: Identification and removal of feature envy bad smells. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 519–523.

[53] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[54] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural program repair by jointly learning to localize and repair. In *7th International Conference on Learning Representations, ICLR 2019.*

[55] Justin D Weisz, Mohit Jain, Narendra Nath Joshi, James Johnson, and Ingrid Lange. 2019. BigBlueBot: teaching strategies for successful human-agent interactions. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*. 448–459.

[56] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.

[57] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, Michael Witbrock, and Vadim Sheinin. 2018. Graph2seq: Graph to sequence learning with attention-based neural networks. *arXiv preprint arXiv:1804.00823* (2018).

[58] Ryan Yen, Jiawen Stefanie Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. 2024. CoLadder: Manipulating Code Generation via Multi-Level Blocks. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 11, 20 pages. doi:10.1145/3654777.3676357

[59] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, Brussels, Belgium, 3911–3921. doi:10.18653/v1/D18-1425

[60] Nicholas Zakas. 2013. ESLint: Pluggable JavaScript linter. https://eslint.org/ Accessed: 2024-10-10.

[61] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 7443–7464. doi:10.18653/v1/2023.acl-long.411

[62] Tianjun Zhang, Fangchen Liu, Justin Wong, Pieter Abbeel, and Joseph E. Gonzalez. 2023. The Wisdom of Hindsight Makes Language Models Better Instruction Followers. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 41414–41428. https://proceedings.mlr.press/v202/zhang23ab.html

[63] Yichi Zhang. 2024. Detecting Code Comment Inconsistencies using LLM and Program Analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 683–685.

[64] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. *Transactions on Machine Learning Research* (2024). https://openreview.net/forum?id=hkNnGqZnpa

[65] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684.