# CFlow: Supporting Semantic Flow Analysis of Students' Code in Programming Problems at Scale

Ashley Ge Zhang
University of Michigan
Ann Arbor, Michigan, USA
gezh@umich.edu

Xiaohang Tang
Virginia Tech
Blacksburg, Virginia, USA
xiaohangtang@vt.edu

Steve Oney
University of Michigan
Ann Arbor, Michigan, USA
soney@umich.edu

Yan Chen
Virginia Tech
Blacksburg, Virginia, USA
ych@vt.edu

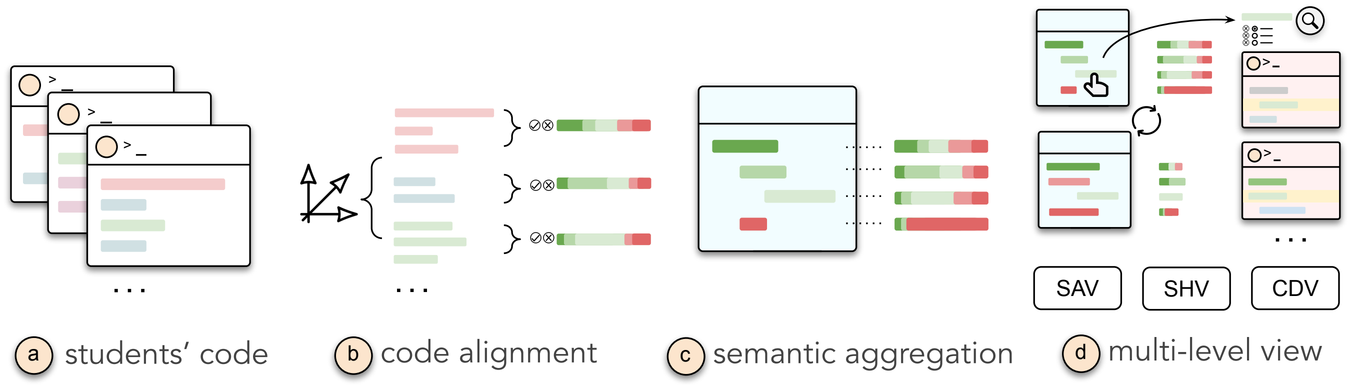


**Figure 1: A system overview of CFlow. (a) Upon collecting student code submissions, (b) CFlow extracts and semantically labels the code statements. (c left) These labels are then clustered and represented in a Semantic Aggregation View (SAV). (c right) Statement correctness is color coded and visualized using a stacked bar histogram, forming the Semantic Histogram View (SHV). (d) Users can filter the views by clicking on labels in the SAV or bars in the SHV. The Code Detailed View (CDV) enables for an in-depth inspection of specific code branches.**

## ABSTRACT

The high demand for computer science education has led to high enrollments, with some introductory courses enrolling thousands of students. In such large courses, it can be overwhelmingly difficult for instructors to understand class-wide problem-solving patterns or issues, which is crucial for improving instruction and addressing important pedagogical challenges. In this paper, we propose a technique and system, *CFlow*, for creating understandable and navigable representations of code at scale. CFlow is able to represent thousands of code samples in a visualization that resembles a single code sample. CFlow creates scalable code representations by (1) clustering individual statements with similar semantic purposes, (2) presenting clustered statements in a way that maintains semantic relationships between statements, (3) representing the correctness of different variations as a histogram, and (4) allowing users to navigate through solutions interactively using semantic filters. With a multi-level view design, users can navigate high-level patterns, and low-level implementations. This is in contrast to prior tools that either limit their focus on isolated statements (and thus discard the surrounding context of those statements) or cluster entire code samples (which can lead to large numbers of clusters—for example, if there are $n$ code features and $m$ implementations of each, there can be $m^n$ clusters). We evaluated the effectiveness of CFlow with a comparison study, found participants using CFlow spent only half the time identifying mistakes and recalled twice as many desired patterns from over 6,000 submissions.

## CCS CONCEPTS

• **Human-centered computing** → **Information visualization**; • **Applied computing** → **Education**; • **Software and its engineering**;

## KEYWORDS

Code Visualization, Programming Education, Code Clustering

**ACM Reference Format:**
Ashley Ge Zhang, Xiaohang Tang, Steve Oney, and Yan Chen. 2024. CFlow: Supporting Semantic Flow Analysis of Students' Code in Programming Problems at Scale. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale (L@S '24), July 18–20, 2024, Atlanta, GA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3657604.3662025

## 1 INTRODUCTION

Understanding student code is crucial for instructors to provide personalized feedback and enhance student learning outcomes. When performing these tasks, instructors often need to: 1) identify *issue-relevant code*, and 2) understand the *underlying reasons* for students' struggles. However, with increasing enrollments in CS courses and the diversity of student codes, these tasks become challenging for two reasons. First, students' errors could span multiple lines in code. A code submission consists of a sequence of code statements (code lines), where each line and the flow from one line to another reflect the student's thoughts on problem solving. An issue might be as simple as a one-line error, such as incorrect dictionary usage, or as complex as a mistake spanning multiple lines, such as improper initialization and modification of variable values. Switching between code syntax and semantics on a large scale is a cognitively demanding task. Second, even with identical syntax structure, two submissions could yield different outputs, and vice versa. For example, students might arrange their if-else statements differently yet produce the same output. This nuanced difference makes it difficult to aggregate a large number of code issues at both the syntax and semantic levels.

Past research has explored this challenge of identifying issues using a variety of methods. For example, OverCode addressed scalability concerns by clustering and visualizing student code submissions [16]. However, they concentrated primarily on specific challenges such as different variable names and statement orders, and focused on the computations of programs, overlooking issues related to high-level misconceptions that might be latent between lines of code. To aid in the discovery of diverse high-level patterns, RunEx enabled instructors to construct runtime and syntax-based search queries with high expressiveness and apply combined filters to code examples [54]. However, RunEx depended on instructors to generate these search queries, which demanded prior knowledge. Additionally, it did not provide guidance on pattern discoverability, leaving users to identify differences between patterns. Other auto-feedback generation approaches have been shown to be promising [53]. However, by excluding instructors from the feedback loop, instructors were uninformed about student challenges. Such detachment could result in instructors not adapting their teaching strategies based on prior student performance.

In this paper, we introduce a novel method that visualizes the flow of code statements to facilitate analyzing students' submissions. Our intuition is that, similar to the narrative flow in an essay, the sequencing of code statements dictates not only the execution order within a program but also its runtime complexity [1]. Within the context of student submissions, this flow sheds light on a student's grasp of the problem, their approach to problem-solving, thought processes, and potential areas of misunderstanding. Thus, effectively visualizing this code flow can uncover dimensions of students' understandings that previous research has not addressed.

However, visualizing code flow presents challenges for two reasons: First, student code submissions often vary significantly, not just in their solutions but also in their structure and semantic meaning. The difficulty lies in aggregating and aligning these diverse submissions. Second, designing a visualization that maintains the code's inherent structure while facilitating flow analysis is difficult, as the same code semantic flow might have different implementations across submissions. Additionally, aggregating code structures on a large scale could make it more cognitively overwhelming for instructors. For instance, by showing the variation within steps across submissions, it could introduce much more complexity at each step of the approaches than simply reading a single code sample. Therefore, visualizations must present flow information in a way that effectively combines code samples and maintains the readability of the original code samples.

To tackle these challenges, we designed CFlow, a system comprising three distinct views (Figure 1d): the *Semantic Aggregation View* (SAV), the *Semantic Histogram View* (SHV), and the *Code Detailed View* (CDV). In essence, CFlow aims to represent semantic flows between distinct value sets and showcases categorical variances. Specifically, CFlow employs a multi-step algorithm where each code line is subjected to a detailed semantic analysis and error check using Large Language Model (LLM). These lines are then vectorized using CodeBert to group them based on similarity. A "common progression" of steps from correct solutions provides a reference structure for mapping all solutions. The resultant structured data is then visualized in a color-coded format, enabling educators to quickly pinpoint student challenges and misconceptions. This method 1) highlights semantic patterns by frequency and accuracy, and 2) simplifies the navigation and comparison of code flows. The core insight of CFlow is to align instructors' analysis of code submissions with the intrinsic characteristics of student code.

To assess the efficacy of CFlow in aiding instructors in identifying issues with student code, we conducted a within-subject experiment involving 16 participants and over 6,000 student code submissions for two programming exercises. To ensure a fair comparison, we designed the baseline system to be the combination of two state-of-the-art systems, OverCode and RunEx [16, 54]. Our findings indicated that, in comparison to the baseline system, CFlow enabled participants to 1) identify targeted misconceptions in half the time used for the baseline, and 2) achieve greater accuracy in their results. CFlow is the first system that bridges the high-level flow among thousands of submissions with specific student errors. By overlaying aggregated semantic data on submissions while retaining their inherent structure, CFlow enhances exploration and navigation capabilities.

Our research underscores the continued need to assist instructors in analyzing large-scale, structurally intricate student data. This research thus contributes:

- A novel visualization approach, and implementation of the approach, that aggregate semantic patterns and code structures, to unearth complex in large-scale, multifaceted code submissions.
- Evidence from an evaluation of CFlow that suggests that CFlow can assist users in identifying a variety of patterns and misconceptions in students' code.

## 2 RELATED WORK

Our work builds on prior work on programming education at scale, code flow analysis, and LLM for programming education.

### 2.1 Mistakes in Introductory Programming

There are primarily three categories of programming knowledge in introductory courses: 1) syntactic knowledge, such as language features and rules, 2) conceptual knowledge, such how programming constructs and concepts work 3) strategic knowledge, which refers to how to apply prior knowledge to solve programming problems [4, 35, 43]. Syntactic mistakes are frequent, but are usually superficial and easy to fix [2, 23]. Conceptual mistakes could lead to significant misconceptions and are relevant to students' thought process [3, 6, 31, 43, 50]. For instance, errors on variable initialization and modification relate to misconceptions on variable scopes [14]. Errors on concepts like conditionals and looping constructs can lead to misconceptions on program execution [17, 48, 49]. Students that lack syntactic and conceptual knowledge could make more strategic mistakes [10, 11], reflecting difficulties in decomposing the programming problem [38, 45]. To identify different types of misconceptions, tools should reveal the various dimensions of students' code. **Therefore, our first design goal (DG1) is to easily understand the semantic flow within students' code.**

## 2.2 Programming Education at Scale

Large courses, such as Massive Open Online Courses (MOOCs), face challenges of maintaining quality and offering personal attention to learners [21]. In the context of programming education, the challenge lies in analyzing students' coding submissions to understand their learning needs and provide tailored feedback. However, the expansive course size and the wide variation among students' coding solutions make it submissions time-consuming and laborious [8, 9, 16, 18, 19, 36, 52].

To address these challenges, various tools have been crafted to provide instructors with an overview of students' code [15, 16, 18, 22, 55]. However, these tools take code submission as single piece and ignore the multifaceted aspect of the program flow, hindering instructors' ability to analyze the flow within thousands of code samples simultaneously. Code search tools could help instructors identify specific coding patterns [33, 34, 39, 42, 54]. Codewebs was a code search engine in educational settings that employed Abstract Syntax Trees (ASTs) and unit test outcomes to match code samples, enabling instructors to index a million code submissions [39]. Codewebs' was limited to filtering by post-execution runtime values. RunEx allowed instructors to construct queries based on runtime and syntax with high expressiveness and to search by combined filters [54]. The downside of code search tool is that instructors need to create queries manually, requiring knowledge of student approaches and challenges. In contrast, CFlow reduces the effort needed for query formulation by analyzing and summarizing code flows across submissions and offering an "available" vocabulary.

## 2.3 Code Flow Analysis in CS Education

Control flow, or the interrelation of statements in a program, is crucial to programming comprehension [1], providing a lens into the programmer's logic, strategy, and misconceptions. Understanding code flow becomes paramount at scale due to the variety of coding solutions from students with diverse backgrounds and thought processes, posing challenges in assessing the correctness of a solution, discerning approaches, identifying misconceptions, and offering tailored feedback [24, 25, 47].

Current educational methods emphasize the process of coding over product, focusing on student crafting their solution as seen in their code flow [51]. Tools like Theseus [29] visualize code segment interactions to spot logical errors. However, their designs struggle with numerous varied solutions in large CS courses.

CFlow overcame this challenge by offering multi-level views of students' solutions with semantic labeling and interactive filtering. CFlow combined elements from Sankey diagrams, adapted for mapping code structures and logic pathways, and histograms for highlighting data distribution and identifying patterns in student code [44, 46]. The combined utility of these visualization techniques in a cohesive system remains largely unexplored in programming education. CFlow's multi-faceted visualization bridges this gap by integrating the flow-centric insights of Sankey diagrams with the succinct data representation of histograms. CFlow simplifies Sankey diagrams to align with how people comprehend code, while preserving interactive flow relationships. CFlow offers a scalable and intuitive method for instructors to navigate numerous student submissions and discern patterns in large CS courses.

## 2.4 LLMs for Programming Education

Large language models (LLMs) [37] have been widely used in programming education for code generation [12, 26], program comprehension [40], language learning [5], and teaching material design [30]. Prior work has designed systems using LLMs to help instructors comprehend students' code more effectively than student-generated explanations [20, 27, 27, 55]. However, LLMs face challenges in conveying complex relationships and structures in code at scale due to their text-based nature. [7]. To tackle the issue, VizProg visualizes students' coding progress as dynamic dots on a 2D map in real time, through clustering vector embeddings of students' solutions using pre-trained language model CodeBERT [55]. VizProg's downside is that it only one level of abstraction, losing structural cues in syntax and semantic meanings. To enhance understanding, a progressive disclosure approach is suggested to gradually provide pieces of information about code that contribute to the overall understanding [41]. **Therefore, our second design goal (DG2) is to seamlessly navigate students' code between high level abstraction and detailed information.** CFlow builds on this by using LLM-created content to aid instructors in processing students' code at scale, employing a hierarchical and "focus + context" design to visualize numerous text information.

## 3 CFLOW

### 3.1 System Design Goals

Derived from the prior literature, we developed three design goals (DG1–DG2) to guide the development of CFlow to help instructors understand and explore student solutions to a programming exercise.

- **DG1: Easily understand the semantic flow within students' code.** Comprehending the semantic flow is essential for grasping code's structure and design. The system should provide an intuitive, concise view that simplifies the semantic flow analysis of students' solutions, thus enhancing overall code comprehension. Visualization of code flow at large scale could be difficult to digest, thus the visualization should align with how human read and comprehend code.
- **DG2: Seamlessly navigate students' code between high level abstraction and detailed information.** When reviewing students' code on a large scale, it's crucial for instructors to constantly dive into specific submissions to ground

their understanding. This allows instructors to identify concrete examples and form contextualized feedback. To support this behavior, the system should offer easy navigation on students' code, thus enabling instructors to switch between "diving in" and "floating up" different samples.

Following the design goals, CFlow's visualization integrates the flow-centric insights of Sankey diagrams with the data distribution conciseness of histograms to present students' code flow. This design can effectively illustrate how data or control moves through different parts of the program. Combined with histograms, the visualization could provide instructors with an overview and reveal bottlenecks in the flow. Furthermore, CFlow provides instructors with a multi-level view of students' code, enhancing their ability to navigate and explore code at scale. It empowers them to discern patterns and identify mistakes, all within the contextual framework of semantic flow analysis. We implemented CFlow as a prototype. In the following sections, we describe CFlow's user interface and the algorithms that were developed to support its features.

## 3.2 CFlow's User Interface

CFlow's user interface is organized into three primary panels: a Semantic Aggregation View (SAV; Figure 2a), a Semantic Histogram View (SHV; Figure 2b), and a Code Detailed View (CDV; Figure 2e). Upon loading student code submissions, CFlow's algorithm (described in detail in Section 3.3) determines the semantic meaning and error information of each code line across all submissions, cluster code lines by their vector similarity, and determined the correctness of the implemented solutions. The SAV displayed code syntax that align with a common semantic flow among all submissions, with color-coding to represent the correctness of each step in the semantic flow. In contrast, the SHV offered a historical view of all code implementations that were clustered as the same step in the SAV, again using color to denote correctness. When a user clicks any of these code lines, the CDV updated its view, showcasing a curated list of code submissions that contained the selected line. Concurrently, both the SAV and SHV updated to reflect this user interaction. In the following sections, we illustrate the details of the system's design.

*3.2.1 The SAV: Semantic Reference Code.* To ease pattern identification (DG1), CFlow presents code in a visualization that is resembles a single code sample (Figure 2a). Nevertheless, this visualization did not originate from a single submission; instead, each line symbolizes a group of similar code lines, like a label or category, drawn from different student submissions. These lines are arranged in the identical sequence as they were found in the students' work. For instance, there was a coherent progression from initializing variables, iterating over variables, to employing conditional statements, mirroring the typical structural approach employed by most students in their code.

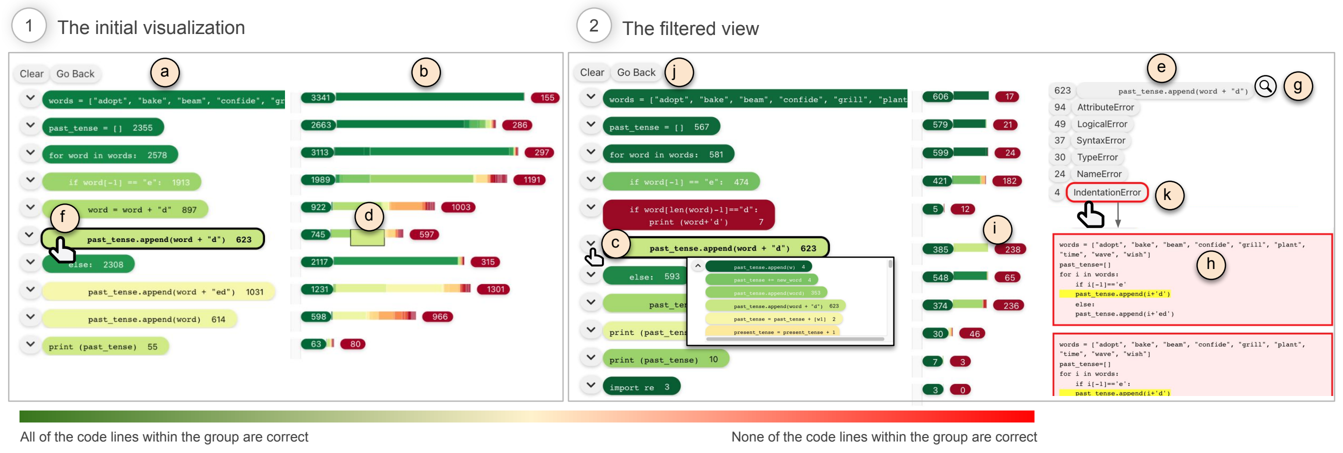
Different from conventional views where correctness is about the whole solution, here we display correctness at code line level. Each line's color-coded representation denotes the correctness of the associated group of code lines. The colors were mapped on a scale from [0, 1] to [Red, Green]. In this scale, Red signified that none of the code lines within the group were correct, whereas Green indicated that all the code lines within the group were correct.

To facilitate exploration of the semantic flow (DG2), CFlow implements clickable elements for all components within the SAV. Users could click on the expand button located to the left of each code line (Figure 2c), revealing similar labels (Figure 2c) that correspond to the same position in the reference code framework. Alternatively, users can directly click on the code line itself (Figure 2f). This action simultaneously updates all system views from the initial visualization of the entire solution set (Figure 2.1) to the filtered view of the subset selected (Figure 2.2). To enhance navigation, users have the option to click the 'Go Back' button to return to a broader view (Figure 2j). The 'Clear' button, on the other hand, allows users to instantly revert to the most expansive view level.

*3.2.2 The SHV: Semantic Code Histogram & Line Correctness.* To assist users in identifying multiple patterns within the data (DG1), CFlow plots a histogram based on the semantic labels detailed in the previous section. The color-coding in this view is an indicator of a collection of code line's correctness. Each stacked bar view is flanked by two numbers representing the count of code lines that are correct and incorrect, respectively. We incorporated the correctness metrics to cater to the needs of users who might be interested in submissions that had an error due to a particular line or chunk of code. An animation (Figure 2d) emphasized the proportion of the current highlighted labels within the overall dataset, thus enhancing navigational ease. Interacting with the SHV was designed to be consistent with the SAV, so clicking on any bar within the SHV triggered a system response just like selecting a label did in the SAV (Figure 2f). Consequently, the SHV (Figure 2i) retained only the clicked bar, while the rest of the histograms adjusted based on their correctness metrics. This interaction paradigm facilitated a nested querying approach. By using a label as a filter, users can delve deeper into specific data subsets, mirroring the nested query functionality in database searches. Here, sequential queries systematically refined the search path, guiding users toward their desired data branch.

*3.2.3 The CDV: Code Search with Error Filters & Semantic Labels.* To assist users in quickly navigating and filtering through students' code by different categories such as error types or specific semantics (DG2), CFlow contains a code search engine equipped with features such as filters (Figure 2g) and context-aware code syntax highlighting (Figure 2h). What sets this design apart from other code search utilities, such as RunEx, is the intricate data-binding that existed across the three primary views of CFlow. As previously discussed, users can initiate a search by clicking on a label or a histogram bar, with the clicked label serving as the search query. This interaction enabled users to continually refine their search with additional click-based queries and layer multiple filters, thus progressively narrowing the list of relevant code submissions.

After uploading code submissions, error type filters were auto-generated using an LLM that was not trained on our dataset (Section 3.3 explains in further detail). The accompanying numbers (Figure 2k) represents the count of submissions that had the corresponding error type at the selected code line. For example, in the scenario depicted in Figure 2e, out of the 623 submissions containing the `past_tense.append(word + “d”)` line, over 200 submissions were identified as having an error at this specific line by the LLM, while the remainder were correct at that line. Users can seamlessly navigate this list to inspect relevant submissions in detail.


1 The initial visualization
Clear
Go Back
a
b
f
d
2 The filtered view
j
c
e
g
i
k
h
623 past_tense.append(word + "d")
94 AttributeError
49 LogicalError
37 SyntaxError
30 TypeError
24 NameError
4 IndentationError
All of the code lines within the group are correct
None of the code lines within the group are correct

```
words = ["adopt", "bake", "beam", "confide", "gr
past_tense = []  2355
for word in words:  2578
    if word[-1] == "e":  1913
        word = word + "d"  897
        past_tense.append(word + "d")  623
    else:  2308
        past_tense.append(word + "ed")  1031
        past_tense.append(word)  614
print (past_tense)  55
```

```
3341 155
2663 286
3113 297
1989 1191
922 1003
745 597
2117 315
1231 1301
598 966
63 80
```

```
words = ["adopt", "bake", "beam", "confide", "grill", "plant
past_tense = []  567
for word in words:  581
    if word[-1] == "e":  474
    if word[len(word)-1]=="d":
      print (word+'d')        7
        past_tense.append(word + "d")  623
    else:  593
        past_te
print (past_tens
print (past_tense)  10
import re  3
```

```
past_tense.append(w)  4
past_tense += new_word  4
past_tense.append(word)  353
past_tense.append(word + "d")  623
past_tense = past_tense + [w1]  2
present_tense = present_tense + 1
```

```
606 17
579 21
599 24
421 182
5 12
385 238
548 65
374 236
30 46
7 3
3 0
```

```
words = ["adopt", "bake", "beam", "confide", "grill", "plant",
"time", "wave", "wish"]
past_tense=[]
for i in words:
    if i[-1]=='e'
    past_tense.append(i+'d')
    else:
    past_tense.append(i+'ed')
```

```
words = ["adopt", "bake", "beam", "confide", "grill", "plant",
"time", "wave", "wish"]
past_tense=[]
for i in words:
    if i[-1]=='e':
    past_tense.append(i+'d')
```

**Figure 2: CFlow allows users to explore the semantic flows of student code submissions at a high level through semantic aggregation. First, users are presented with an overview of the entire set of solutions (1), including the SAV (a) and the SHV (b). They can then click on individual code lines (f) to progressively explore the details of specific implementations. The visualization will update to focus on a smaller subset of solutions, displaying the aggregated flow and distributions of the selected set only (2). Users can inspect details of the flow at individual code level in the CDV (e). CFlow offers a breakdown of types of errors within that group (k), and detailed solutions with context-aware highlighting (h).**

## 3.3 CFlow's Algorithm

To generate the aforementioned visualization, CFlow requires detailed information about correctness of students' code at the line level. To effectively visualize the information, CFlow segments the code into various components based on the semantic flow and then group these segments according to similarity. Specifically, CFlow produces its visualization using four primary stages (Figure 3):

- **Stage 1:** Identifying and tagging the steps required to solve a problem
- **Stage 2:** Grouping and aligning code lines across code samples
- **Stage 3:** Identifying semantic, syntactic, and runtime errors
- **Stage 4:** Clustering the grouped results

We will describe each stage in the sub-subsections below.

### 3.3.1 *Stage 1: Identifying and tagging the steps required to solve a problem.*

First, CFlow identifies the common steps across the code samples (Figure 3.1). Doing this requires understanding the *semantic meaning* of these code samples. For example, "`if x != 5:`" is functionally identical to "`if not (2+3 == i):`" if `x` and `i` serve the same purpose in code. Further, many code samples contain minor syntax errors (like omitting a parentheses in "`if not 2+3 == i):`") that should be classified as semantically similar.

Thus, for each code sample (which we will denote as $c_i$ to represent code sample $i$) CFlow first 'normalizes' the code using text-based normalization techniques. This includes removing extra whitespace, stripping code comments, identifying variable names and remapping them to be consistent across code samples, and removing `print()` function calls. We will refer to the 'normalized' version of code sample $c_i$ as "norm $(c_i)$". CFlow then divides the normalized code sample into lines; we will denote the first line of $c_i$ as $c_i^1$, the second line as $c_i^2$, etc. CFlow then uses CodeBERT [13] to vectorize of norm $(c_i)$, which we will denote as $\text{vec}(\text{norm}(c_i)^j) \in \mathbb{R}^{768}$ for line $j$ of code sample $i$. We will use $v_i^j$ as shorthand for $\text{vec}(\text{norm}(c_i)^j)$. This vectorized representation ($v_i^j$) captures the *semantic* meaning of the code line and is resilient to small variations and errors. It is also contextualized in the larger code sample to capture what $\text{norm}(c_i)^j$ means *in context*.

Next, CFlow uses agglomerative clustering[1] to cluster all code lines across samples $\{v_1^1, v_1^2, ..., v_2^1, v_2^2, ...\}$ to group similar lines of code. Each line of code is placed into one cluster. We will use $\tau_i^j$ to represent the ID of the cluster that line $v_i^j$ is placed in. The cluster ID $\tau_i^j$ is then used as a *tag* for each line. At the conclusion of stage 1, each line in each code sample has a tag ($\tau_i^j$) that represents its semantic meaning. Every line of code with the same tag $\tau$ is semantically similar.

### 3.3.2 *Stage 2: Grouping and aligning code lines across code samples.*

In stage 1, CFlow created semantically meaningful labels ($\tau_i^j$ is the label for line $j$ of code sample $i$). Although these labels capture the semantic meaning of every given code line, they were created independent of each line's location in the larger code sample, a necessary component for CFlow's visualization to represent many code samples coherently. To do this CFlow then *aligns* lines of code across samples (Figure 3.2).

As a first step for alignment, CFlow identifies a "canonical progression" —a set of steps that represents the "average" correct solution. To identify these steps, CFlow first identifies which code samples are *correct*, using unit tests. Across each *correct* solution $\text{norm}(c_i)$, CFlow identifies the most common tag for each individual line across code samples. We denote this progression as $\mathrm{T} = (\mathrm{T}_1, \mathrm{T}_2, ...)$. This means, $\mathrm{T}_1$ is the most common value of the tag for the first line ($\tau_i^1$) among correct solutions, $\mathrm{T}_2$ is the most common value for $\tau_i^2$, etc. The length of the canonical progression $\mathrm{T}$ depends on a pre-set parameter that determines the minimum

[1]https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html

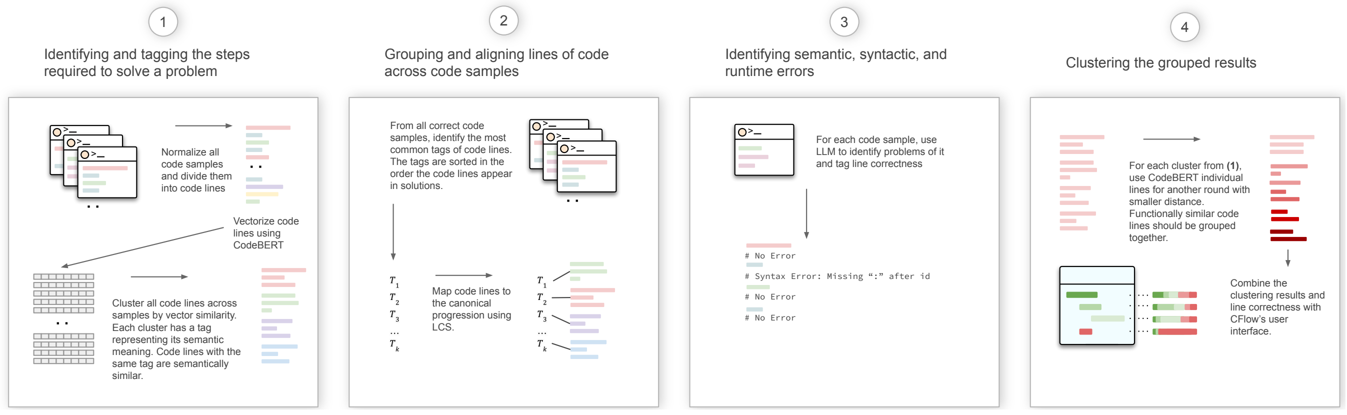

1
Identifying and tagging the steps required to solve a problem
Normalize all code samples and divide them into code lines
Vectorize code lines using CodeBERT
Cluster all code lines across samples by vector similarity. Each cluster has a tag representing its semantic meaning. Code lines with the same tag are semantically similar.
2
Grouping and aligning lines of code across code samples
From all correct code samples, identify the most common tags of code lines. The tags are sorted in the order the code lines appear in solutions.
$T_1$ $T_2$ $T_3$ … $T_k$
Map code lines to the canonical progression using LCS.
3
Identifying semantic, syntactic, and runtime errors
For each code sample, use LLM to identify problems of it and tag line correctness
# No Error
# Syntax Error: Missing ":" after id
# No Error
# No Error
4
Clustering the grouped results
For each cluster from (1), use CodeBERT individual lines for another round with smaller distance. Functionally similar code lines should be grouped together.
Combine the clustering results and line correctness with CFlow's user interface.

**Figure 3: CFlow's algorithm. To generate the results required in CFlow's user interface, CFlow's algorithm include four primary stages: (1) identifying and tagging the steps required to solve a problem, (2) grouping and aligning lines of code across code samples, (3) identifying semantic, syntactic, and runtime errors, and (4) clustering the grouped results.**

agreement across solutions, stopping once it reaches a line number where tags are not sufficiently consistent. Finally, CFlow computes the longest common subsequence (LCS) between each code sample's tags ($\tau_i^1, \tau_i^2, \ldots$) and ($\mathrm{T}_1, \mathrm{T}_2, \ldots$).

One assumption made by this approach is that there is one representative "average" progression T—most correct solutions use a similar set of steps. We adopt this approach since submissions for introductory programming problems are short and follow a few dominant progressions. Future iterations of this algorithm could address this limitation by producing a value of T for each approach.

*3.3.3 Stage 3: Identifying semantic, syntactic, and runtime errors.* An important aspect of CFlow's visualization is that it helps users understand *common problems* and errors that learners face. Testing code samples against unit tests can identify which code samples contain errors but does not help identify which specific parts of the code samples are problematic. Further, even identifying lines that result in runtime or syntax errors might not help identify the true actual source of a given error. For example, a runtime error might occur on a correct line if it references a variable that was incorrectly set on a line before it.

Thus, to identify lines of code with errors more accurately, CFlow passes each code sample $c_i$ through an LLM (ChatGPT-3.5) that is prompted to identify problems in the code (Figure 3.3). For each line, this LLM identifies whether the line is (1) correct or contains (2) a semantic error, (3) a syntax error, or (4) a runtime error. We denote this information for line $j$ of code sample $i$ as $E_i^j$. The correctness at the code line level was determined by GPT3.5-turbo. The prompt used for this determination can be found in appendix (Section D).

*3.3.4 Stage 4: Clustering the Grouped Results.* After CFlow identifies a canonical progression (T) and maps each line across code samples to that progression (stage 2), it then uses agglomerative clustering to cluster every lines of code within each tag $\tau$ (Figure 3.4). Recall that the tag for line $j$ of code sample $i$ ($\tau_i^j$) was computed by clustering vectorized code lines ($v_i^j$). As a result, code lines that are functionally identical (such as "`if x != 5:`" and "`if not 2+3 == i):`") should end up with the same tag $\tau$. Finally, for every line, CFlow then performs another round of clustering of code line vectors $v$ *within* grouped lines with the same tag $\tau$, with a smaller maximum distance metric. These clusters are then used to group solutions within the CFlow UI and are combined with the error information generated in step 3.

## 3.4 Evaluation of CFlow's Algorithm

Using LLM-powered approaches could raise certain concerns, including low transparency, lack of control, and issues of trust [28]. To determine if the performance of the LLM impacts CFlow's effectiveness, we conducted a comparative study assessing the accuracy and reliability of our approach, against human judgment. Specifically, we analyzed how well the LLM (GPT-3.5-turbo[2]) used in CFlow identifies issues within code samples (details in Section 3.3.3). We did not evaluate the other LLM-powered component in CFlow, CodeBERT, as its efficacy has been extensively studied and validated in previous research[13, 32, 56], including tasks on automated program repair, program clustering, and software defect prediction.

CFlow used LLM to label each line of a code sample as either (1) correct, (2) having semantic error(s), (3) having syntax error(s), or (4) having runtime error(s). We evaluated the LLM's performance in identifying the semantic errors, by comparing that of expert human labeler. However, expert human labeler could have different opinions on which lines are incorrect in a submission. Some people would only take lines that need edits as incorrect, while other people would take the whole program construct involving the erroneous lines as incorrect. Instead of designing a rubric of a single standard and label ground truth, we recruited two experienced Python programmer to label 50 solutions to 5 programming exercises, selected code lines that are incorrect, and compared the results to LLM's outputs. All the solutions in the dataset had semantic errors.

In the study, the first participant (L1) annotated the dataset by marking the code lines with issues. L1 was then asked to review the LLM's annotations and update their original labels if necessary. The second participant (L2) received both L1's revised annotations and the LLM's annotations, but was not informed about the source of each label set. L2's task involved comparing these two annotation sets and choosing one of four options: (1) version 1 is correct, (2)

[2]https://platform.openai.com/docs/models/gpt-3-5

version 2 is correct, (3) both versions are correct, or (4) both versions are incorrect. Beyond evaluating LLM's performance, this study design accounts for potential biases in human labeling (e.g., level of specificity) and enables us to evaluate the representativeness of the LLM's labels (e.g., whether the incorrect labels are made up).

To evaluate the level of agreement, we considered labels marking specific lines with issues and labels applied to the entire code block containing that line as equivalent. This approach is justified as human labelers might annotate lines with issues at varying levels of specificity, yet still indicate the same error. For instance, in a solution with an incorrect combination of condition and content in an if-else statement, labeling just the condition and labeling on the entire if-else statement were considered the same. Our analysis of the labeling results showed that the two participants agreed on 96% of the lines. L1 incorrectly labeled two solutions, which L2 subsequently corrected. LLM's outputs agreed with L1 and L2 on 80% and 90% of the lines, respectively. However, upon closer examination of the specific solutions and error messages generated by the LLM, we discovered that LLM actually achieved 96% and complete agreement with L1 and L2, respectively. For example, when incorrect solutions missed variables that are required by the exercise, L1 labeled the lines where the variables were initialized, whereas LLM labeled the lines where the variables were used. Therefore, we believe that CFlow's algorithm is reliable for educational settings. We will discuss this with more details in Section 5.3.

## 4 USER STUDY

A within-subject study was conducted to evaluate CFlow's efficacy in supporting instructors to understand student code submissions at scale. The baseline tool, combined the core functionalities of two state-of-the-art research tools, RunEx and OverCode, that were designed with a similar goal to CFlow [16, 54]. In the study, participants used either the baseline system or CFlow to answer quiz questions on students' errors on patterns and logic consistency. The Institutional Review Board (IRB) approved the study, ensuring adherence to ethical standards and participant safety.

### 4.1 Method

*4.1.1 Participants.* Because CFlow's end users would be programming instructors, we reached out to senior students from *[redacted for anonymity]* who had experience teaching Python programming courses. During a screening session, participants indicated their prior experience teaching and using Python. Teaching assistants and senior students with at least 3 years of experience were invited to participate in the study. Given that participants with teaching experience might anticipate certain student mistakes, we also included non-teaching participants that are senior students and experienced in Python. In total 16 participants were recruited (i.e., 4 self-identified as male, 12 as female) and their experience with Python programming ranged from 2 years to 8 years, with 14 participants having previously taught programming courses in Python and the other 2 being senior students that are experienced in Python.

*4.1.2 Study Systems.* Since there is no widely used commercial tools that identify students' mistakes and approaches in code to compare with, we designed the baseline system as a combination of RunEx [54] and OverCode [16]. Both RunEx and OverCode are the-state-of-art systems for viewing students solutions. Their user studies showed benefits from various aspects - RunEx helps users identify specific code patterns with higher accuracy and expressiveness [54], while OverCode allows teachers to quickly develop a high-level view of students' understanding and misconceptions, and to provide relevant feedback to students [16]. However, OverCode's limitation was in analyzing mistakes within students' code, as it required solutions to be free of syntax errors, and categorized solutions that were syntax-error-free but failed the unit tests in their own distinct cluster. To fill this missing part of OverCode and ensure a fair comparison, we complemented OverCode's clustering results with RunEx [54], a code search tool designed for programming education. RunEx enables users to explore class-wide patterns within a large volume of student code by augmenting regular expressions with runtime values for enhanced functionality. During the user study, we used the user interface of RunEx and display OverCode's clustering results in it. Both CFlow and the baseline systems were implemented as standalone websites.

Details of the baseline system's user interface can be found in Appendix C. To ensure that participants fully understand how to use the system, we spent 20 minutes training participants on using it and asked participants to try out the system with test tasks before the formal tasks. It should be noted that while CFlow offers line correctness, which the baseline system lacks, this is a unique component of our paper. Apart from this, both systems provide similar information. The primary distinction lies in the methods of information presentation. Consequently, we consider our study design to be a fair basis for comparison.

*4.1.3 Programming Problems and Students' Solutions.* To ensure the authenticity of the data used in the study, we collected data from a large introductory programming course at *[redacted for anonymity]*. This data consisted of students' solutions to two distinct programming problems assigned in the course, completed on their own time. The data were collected from an interactive Python textbook used by the course. The data contain genuine examples of mistakes and common patterns students when approaching the problems. To maintain comparability across the systems, we selected one programming exercise from the dataset for each system that had a comparable level of complexity. Specifically, to complete these two exercises, students need to iterate through the provided list, perform string comparison, and then modify the value in other list or dictionary variables.

Exercise 1 (E1): For each word in `words`, add 'd' to the end of the word if it ends in 'e' to make it past tense. Otherwise, add 'ed' to make it past tense. Save these past tense words to a list `past_tense`.

Exercise 2 (E2): Given a string, return a variable `counts`, where the keys are letters in the string, the values are how many times each letter appears in the string.

E1 and E2 each had 3496 and 3249 Python code examples. The solutions varied from 3 lines to 15 lines of code. We checked each submission to ensure that it did not contain any identifying information or present any privacy concerns and anonymized appropriately.

*4.1.4 Study Setup.* The study was conducted remotely using Zoom. Participants joined two sessions, one that used the baseline system and one that used CFlow. In the first session, participants all worked on quiz questions about E1 and in the second session, participants all worked on quiz questions about E2. System order was counterbalanced and we provided 20 minutes of training on how to use the system in each session. After training, participants had 20 minutes to answer quiz questions about students' mistakes and approaches using the assigned system. After finishing the quiz questions, participants were asked to complete a survey about their experience using the system. After the second session (with the same procedure but using the other system), we conducted a reflective interview to compare the two systems. Each participant was compensated with a $25 USD Amazon Gift Card for the completion of each session.

*4.1.5 Question Design.* To ensure a fair comparison between CFlow and the baseline system, we carefully designed our quiz questions to include both multiple-choice and open-ended types. For the multiple-choice questions, we framed them in ways like "Find how many students made this mistake or pattern in their code. Select the closest number from the options below." This approach helped maintain fairness for both systems. On the other hand, the open-ended questions asked participants to identify as many common mistakes as they could find in students' code using each system. To establish correct answers for the quiz questions, one of our team members compiled a list of accurate responses. Importantly, we didn't rely directly on the numbers generated by LLMs when creating the answer options. To ensure objectivity, a team member used both CFlow and the baseline system to perform the tasks and designed the correct options to closely match both conditions. This approach was crucial for maintaining impartiality in our evaluation.

*4.1.6 Data Collection and Metrics.* During the study, we recorded participants' screens as they performed the tasks, as well as their responses to the quiz questions, their audio think-aloud processes, and their answers to the post-study survey and follow-up interview. For each session, one member of the research team was present.

We used two metrics to evaluate participants' answers to the quiz questions. For the multiple-choice questions, we calculated

$$\frac{\text{\# matched answers}}{max(\text{total \# correct answers, total \# selected answers})} \tag{1}$$

to work out their accuracy for the questions. For a quiz questions that have four options A, B, C, and D, where A and B are correct, if a participant selected A, C, and D, the accuracy is 1 / max(2, 3), which is 0.33. For the open-ended questions, we coded the valid mistakes participants found during the study based on a list of existing mistakes generated by the researcher.

We created a list of code scheme of behaviors observed within the screen recordings. We also coded the screen recordings to analyze the time spent on the multiple-choice questions in the quiz. For the open-ended questions in the quiz, we analyzed the screen recordings to understand how participants interacted with the tool to perform the tasks. For the post-study survey and the follow-up interview data, one member of the research team used a thematic analysis to identify recurring themes and insights in the data. We used a paired t-test for the statistical analysis.

## 4.2 Results

We present results of the user study below. We mainly focused on participants' performance in the quiz questions.

*4.2.1 Accuracy of Multiple-Choice Questions in the Quiz.* As mentioned in Section 4.1.6, we calculated Equation 1 to work out their accuracy for the multiple-choice questions in the quiz. We found that participants using CFlow ($\mu$ = 93.02, $\sigma$ = 0.06) had accuracy significantly higher than using the baseline system ($\mu$ = 52.40, $\sigma$ = 0.18, $p$ < 0.0001).

*4.2.2 Time to Identify Patterns and Mistakes in Code.* To understand the influence of the systems on the duration of time participants spent to answer the multiple choice questions, we computed how long it took to answer those questions that related to student mistakes and patterns. Participants using CFlow ($\mu$ = 499.06 seconds, $\sigma$ = 201.47) completed the questions significantly faster then using the baseline system ($\mu$ = 817.50 seconds, $\sigma$ = 264.85, $p$ < 0.001). One participant mentioned that code clustering at line level visualized the errors clearly in CFlow (P15).

*4.2.3 Number of Valid Mistakes.* As mentioned in Section 4.1.6, we coded the valid mistakes participants found in the open-ended questions in the quiz. The results showed that participants found significantly more valid mistakes using CFlow ($\mu$ = 4.81, $\sigma$ = 1.91 than the baseline system ($\mu$ = 2.375, $\sigma$ = 1.58, $p$ < 0.001). We listed the mistakes participants identified in Appendix (Table 1). When using the baseline system, 5 participants only identified syntax errors and described them in a general way, such as "Type Error" and "Name Error", and 3 participants did not list any mistakes due to time constraints that prevented them from thoroughly reviewing the majority of the solutions. Conversely, when using CFlow, all 16 participants were able to comprehensively explore the entire solution set and provide detailed descriptions of the identified mistakes by pointing out the areas that made the solution incorrect.

## 4.3 Findings

We analyzed participants' answers to the post-study survey (as shown in Table 2) and their interview responses. To understand CFlow's usability benefits or issues, we also analyzed participants' interaction and behavior patterns from the video.

*4.3.1 Participants found CFlow helpful in understanding students' code.* On a scale of 7 where 1 is completely disagree and 7 is completely agree, participants disagreed that CFlow is less helpful in understanding students' approaches than the baseline system ($\mu$ = 3.38, $\sigma$ = 1.80). 9 participants (P1-4, P7-10, P13) expressed that CFlow was more helpful in understanding students' approaches than the baseline system because of the color-coded histogram (P13) and the overall semantic flow (P8, P13). CFlow highlighted the majority of students and allowed users to layer the filters on code submissions (P13). 1 participant thought CFlow and the baseline system were comparable in understanding approaches (P14).

Participants also found CFlow more helpful in understanding the code structure than the baseline system. On a scale of 7 where 1 is completely disagree and 7 is completely agree, participants disagreed that CFlow is less helpful in understanding the code structure information than the baseline system ($\mu$ = 2.94, $\sigma$ = 1.48). In the interview responses, 11 participants expressed that they preferred CFlow in understanding the code structure (P1-5, P7-10, P13, P16). CFlow presented a concise view that elucidates the primary steps students take in their code, thereby showing how most students structure their code to solve a programming exercise (P1-2, P5, P7-8). In contrast, the baseline system required users to read every individual code solution and search for a specific pattern of a few lines of code, taking more effort (P2-3, P7, P10-11).

However, some participants also mentioned that the grid view in the baseline system is simple and straightforward, and had lower learning curve for them (P5-6, P11-12, P15-16)

*4.3.2 CFlow fostered an exploratory and brainstorming approach to understand students' code.* To better understand how participants use both systems to comprehend students' code, we analyzed their interaction with the systems. We found that participants had very different strategies in answering questions across the two system.

In the baseline system, participants had two strategies, 1) random browsing and 2) active searching without guidance. Since the baseline system directly displayed clusters derived from OverCode, each incorrect solution was categorized into its own cluster. Without categorized information of mistakes, some participants tried to look through all the incorrect solutions and reported mistakes they encountered during the process (P1-4, P7, P9, P15). Other participants would first come up with some coding patterns and search for them, and then look for other unseen patterns by filtering out what has been seen (P10, P13). Usually what they come up with is a correct pattern instead of incorrect patterns. Due to the lack of guidance, browsing mistakes using the baseline system largely depended on what users had in mind and required users to have an expectation about potential mistakes students would have.

In contrast, when using CFlow, all participants first looked at the semantic-label abstractions and the color-coded histogram to locate where most mistakes were and what the common mistakes were. With the guidance provided in CFlow, they then selectively dug deeper into the details to reason about students' misconceptions.

The difference indicated that CFlow and the baseline system each better fits different settings. participants expressed a preference for using CFlow when identifying common mistakes and exploring students' code on a larger scale, particularly in extensive programming classes with hundreds or thousands of students (P2, P7-8, P10-11, P13). The efficient navigation capabilities in CFlow enables swift exploration of students' code (P2, P5-7), while the baseline system required participants to read each code solution individually (P1, P3, P5, P7, P9-10, P13, P15). Some participants expressed a preference for the baseline system during smaller sessions when seeking specific patterns (P1, P8, P10, P13).

Furthermore, we found that the ability to explore and brainstorm students' code might be beneficial to people with little teaching experience. We analyzed the mistakes listed by two participants who did not have teaching experience. P10 did not have any time to list common mistakes when using the baseline system and listed 4 mistakes with details when using CFlow. P6 listed only 4 general types of syntax errors when using the baseline system, while in CFlow listed 7 mistakes with details including semantic errors. This being said, with CFlow's guidance in browsing mistakes, even participants with limited teaching experience and few expectations of students' code could effectively identify common mistakes.

*4.3.3 CFlow help participants navigate students' solutions with less context switching.* Based on our observation, we found that CFlow takes participants less context switch to understand mistakes within thousands of code solutions. In CFlow, participants can quickly understand errors of the whole classroom by looking at the dominant patterns in SAV and SHV, and inspecting on individual solutions by clicking to view CDV. Participants noted that they could effortlessly locate the information they desired and seamlessly switch between abstraction and finer details (P1-3, P5, P9-10, P12-13, P15).

Participants were asked to identify common implementations used by students. When using the baseline system, 4 out of 8 participants generated search queries, filtered out relevant portions of the dataset, and repeated this process until they no longer observed common implementations. On the other hand, 4 participants listed a single implementation and moved on to other questions. However, when utilizing CFlow, all participants initially located a specific step within the semantic flow, expanded the step to view all implementations, and identified common implementations by referring to the numerical counts alongside them in the histogram.

When locating the steps that have the most mistakes in CFlow, all participants directly used the color-code histogram. They looked at both the dominant color of the histogram and the point where it transitioned from green to red. They then inferred that the step where the majority of it was light green should have had more mistakes than those where the majority of it was dark green.

## 5 DISCUSSION

In this section, we (1) discuss how CFlow contributes to the body of work in visualizing students' code at scale, (2) discuss the role of LLM in CFlow, and (3) discuss CFlow's limitation.

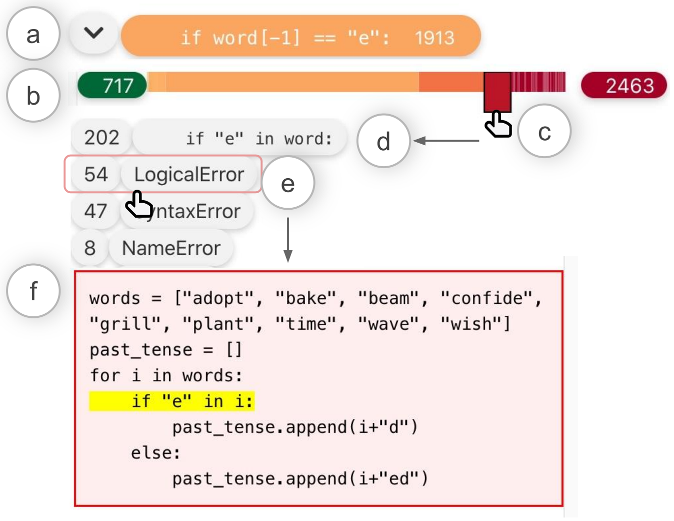
### 5.1 Connecting Individual Code Solutions with the Broad Context of the Whole Dataset

Our findings highlight the transformative impact of CFlow. By compacting thousands of student solutions into a singular, coherent view that traces a shared semantic trajectory, CFlow unlocked how participants interpreted and interacted with the data from two perspectives: they could immerse themselves in the specifics of individual solutions, yet never lose sight of the grand tapestry of submissions. The color-coded error information on CFlow's semantic code view enriched this experience, illuminating subtle line-level differences and fostering a more nuanced understanding.

In contrast, the baseline system required participants to inspect each incorrect solution individually to identify errors and search for specific patterns. This approach was less integrated than the aggregated view offered by CFlow, which naturally linked individual solutions to the broader dataset (as noted by P1-4, P7, P9, P15). While tools like OverCode reduced the number of solutions instructors had to review, their grid layouts often isolated solutions. In comparison, CFlow effectively demonstrated the connections between individual solutions.

### 5.2 Bridging Abstraction and Code Details

When viewing code solutions at scale, users often toggle between two distinct levels of abstraction: the high-level overview of the entire solution set and the low-level details of individual solutions. Our user study revealed that CFlow introduces an additional intermediate level of information, effectively bridging the gap between the high-level abstractions and the details of code. For instance,



```
if word[-1] == "e":  1913
```

```
if "e" in word:
```

```
words = ["adopt", "bake", "beam", "confide",
"grill", "plant", "time", "wave", "wish"]
past_tense = []
for i in words:
    if "e" in i:
        past_tense.append(i+"d")
    else:
        past_tense.append(i+"ed")
```

**Figure 4: An example of how CFlow looks like without LLM determining line correctness. (a) is a collection of code lines that check the end of a word, and (b) is the correctness histogram. Upon selecting the prominent red block (c), users can view an example that incorrectly check the end of a word (d). By clicking on "LogicalError" (e), users are then able to explore detailed solutions (f).**

by looking at the initial visualization without interaction, one participant noticed a reduction in the histogram's size after variable initialization and for loops. This indicated that students might encounter more challenges when writing conditional statements, as many submissions ended at the content inside the for loops. Furthermore, we found that participants could identify common approaches through the preview of the code lines associated with the visualization, without delving into the exact code content.

CFlow offers a novel approach for effectively viewing extensive code data. While many existing tools have introduced innovative presentations, such as VizProg's dynamic dots on a 2D map [55], they often fall short in conveying semantic meaning through their visual positioning. This absence forces instructors to manually select specific areas and delve into the raw code to understand the content. CFlow pioneers a design that seamlessly integrates code lines into a semantic view, replete with visual cues. This approach underscores the advantages of presenting multiple levels of code representation, effectively bridging the gap seen in previous tools.

## 5.3 LLM's Role in CFlow

CFlow uses LLM to generate line-level error information for student code. We discuss LLM's role in CFlow from two angles: 1) the accuracy of LLM, and 2) CFlow's dependency on LLMs.

CFlow is primarily designed to provide instructors with an overarching view of the entire solution set to facilitate exploration and analysis. Therefore, the LLM does not need to produce perfectly accurate error information for each code line. Our user study revealed that participants primarily use the color distribution in the histogram view to pinpoint steps where most students have mistakes, and then examine the specific code lines for a deeper understanding of these errors. Even if disregarding thel ine correctness information from LLM and labeling all lines from an incorrect solution as erroneous, CFlow's visual design could provide valuable insights into students' mistakes. For instance, Figure 4 is an example without LLM's outputs, where instructors can still observe code lines grouped by semantic meanings (Figure 4a) and color-coded for correctness (Figure 4b). After clicking a prominent red block (Figure 4c), instructors can see that the cluster represents the code line "`if "e" in word:`" (Figure 4d), which incorrectly check the end of a word. Upon clicking "LogicalError" (Figure 4e), instructors can then explore detailed solutions (Figure 4f). Despite the less pronounced color difference compared to the original design, instructors are still able to get useful information and navigate students' solutions.

While CFlow relies on LLM for locating semantic errors, we argue that LLM can be replaced in CFlow. Any tool capable of pinpointing semantic errors could potentially replace LLM in CFlow. Alternatives might include a smaller language model or an earlier version of ChatGPT. Moreover, implementing more targeted test cases could help in detecting the precise lines with errors. We chose ChatGPT-3.5 for its widespread accessibility and ease of use. For real classroom deployment, instructors have the flexibility to substitute LLM with any other tool that effectively locates semantic errors.

## 5.4 Limitations and Future Work

One challenge is the cost of using LLMs and their applicability in real-time settings. CFlow used an LLM for post-hoc code analysis, but applying this approach in real time demands both financial and time resources heavily. Future work should seek alternatives that balance computational cost with real-time needs. Another limitation is that the user study tested only two programming problems, which may not reflect the model's performance across different topics and languages. Future studies should investigate a wider variety of problems.

## 6 CONCLUSION

In this research, we addressed the challenges faced by instructors when analyzing large numbers of varied code submissions from students. We introduced CFlow, a novel system that uses semantic labeling and code structure to visualize the semantic flow of students' submissions. By abstracting code statements based on their meaning while maintaining structure, CFlow offers a comprehensive view that simplifies navigation and comparison of code flows. Our evaluation showed that CFlow allows educators to analyze these submissions more effectively than traditional methods. Participants found it easier to explore patterns and understand code structure with CFlow. This work highlights the potential for improving teaching methods by better understanding student submissions and delivering targeted feedback at scale.

## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under DUE 1915515.

## REFERENCES

[1] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.

[2] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM technical symposium on computer science education*. 522–527.

[3] Piraye Bayman and Richard E Mayer. 1983. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Commun. ACM* 26, 9 (1983), 677–679.

[4] Piraye Bayman and Richard E Mayer. 1988. Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology* 80, 3 (1988), 291.

[5] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard-Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 500–506.

[6] José Juan Canas, Maria Teresa Bajo, and Pilar Gonzalvo. 1994. Mental models and computer programming. *International Journal of Human-Computer Studies* 40, 5 (1994), 795–811.

[7] Kuo-En Chang, Yao-Ting Sung, and Ine-Dai Chen. 2002. The effect of concept mapping to enhance text comprehension and summarization. *The Journal of Experimental Education* 71, 1 (2002), 5–23.

[8] Yan Chen, Jaylin Herskovitz, Gabriel Matute, April Wang, Sang Won Lee, Walter S Lasecki, and Steve Oney. 2020. EdCode: Towards Personalized Support at Scale for Remote Assistance in CS Education. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5.

[9] Yan Chen, Walter S Lasecki, and Tao Dong. 2021. Towards supporting programming education at scale via live streaming. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW3 (2021), 1–19.

[10] Michael De Raadt. 2008. *Teaching programming strategies explicitly to novice programmers*. Ph. D. Dissertation. University of Southern Queensland.

[11] Alireza Ebrahimi. 1994. Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies* 41, 4 (1994), 457–480.

[12] Anna Eckerdal and Michael Thuné. 2005. Novice Java programmers' conceptions of" object" and" class", and variation theory. *ACM SIGCSE Bulletin* 37, 3 (2005), 89–93.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[14] Ann E Fleury. 1991. Parameter passing: The rules the students construct. *ACM SIGCSE Bulletin* 23, 1 (1991), 283–286.

[15] Matheus Gaudencio, Ayla Dantas, and Dalton DS Guerrero. 2014. Can computers compare student code solutions as well as teachers?. In *Proceedings of the 45th ACM technical symposium on Computer science education*. 21–26.

[16] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 1–35.

[17] TRG Green. 1977. Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology* 50, 2 (1977), 93–109.

[18] Philip J Guo. 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 599–608.

[19] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 89–98.

[20] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the responses of large language models to beginner programmers' help requests. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. 93–105.

[21] Khe Foon Hew and Wing Sum Cheung. 2014. Students' and instructors' use of massive open online courses (MOOCs): Motivations and challenges. *Educational research review* 12 (2014), 45–58.

[22] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. 2013. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED 2013 Workshops Proceedings Volume*, Vol. 25. Citeseer.

[23] James Jackson, Michael Cobb, and Curtis Carver. 2005. Identifying top Java errors for novice programmers. In *Proceedings frontiers in education 35th annual conference*. IEEE, T4C–T4C.

[24] Lisa C Kaczmarczyk, Elizabeth R Petrick, J Philip East, and Geoffrey L Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*. 107–111.

[25] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 119–125.

[26] Natalie Kiesler and Daniel Schiffner. 2023. Large Language Models in Introductory Programming Education: ChatGPT's Performance and Implications for Assessments. *arXiv preprint arXiv:2308.08572* (2023).

[27] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing code explanations created by students and large language models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 124–130.

[28] Q Vera Liao and Jennifer Wortman Vaughan. 2023. AI Transparency in the Age of LLMs: A Human-Centered Research Roadmap. *arXiv preprint arXiv:2306.01941* (2023).

[29] Tom Lieber, Joel R Brandt, and Rob C Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2481–2490.

[30] Xinyi Lu, Simin Fan, Jessica Houghton, Lu Wang, and Xu Wang. 2023. ReadingQuizMaker: A Human-NLP Collaborative System that Supports Instructors to Design High-Quality Reading Quiz Questions. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–18.

[31] Linxiao Ma. 2007. *Investigating and improving novice programmers' mental models of programming concepts*. Ph. D. Dissertation. Citeseer.

[32] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 505–509.

[33] George Mathew, Chris Parnin, and Kathryn T Stolee. 2020. Slacc: Simion-based language agnostic code clones. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 210–221.

[34] George Mathew and Kathryn T Stolee. 2021. Cross-language code search using static and dynamic analyses. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 205–217.

[35] Tanya J McGill and Simone E Volet. 1997. A conceptual framework for analyzing students' knowledge of programming. *Journal of research on Computing in Education* 29, 3 (1997), 276–297.

[36] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. 2018. A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education* 62, 2 (2018), 77–90.

[37] Cade Metz. 2021. AI can now write its own computer code. That's good news for humans. *The New York Times* 9 (2021).

[38] Orna Muller. 2005. Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the first international workshop on Computing education research*. 57–67.

[39] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web*. 491–502.

[40] Maciej Pankiewicz and Ryan S Baker. 2023. Large Language Models (GPT) for automating feedback on programming assignments. *arXiv preprint arXiv:2307.00150* (2023).

[41] Roy D Pea. 1987. User centered system design: new perspectives on human-computer interaction. *Journal educational computing research* 3, 1 (1987), 129–134.

[42] Andy Podgurski and Lynn Pierce. 1993. Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, 3 (1993), 286–303.

[43] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–24.

[44] Patrick Riehmann, Manfred Hanfler, and Bernd Froehlich. 2005. Interactive sankey diagrams. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005*. IEEE, 233–240.

[45] Anthony Robins, Patricia Haden, and Sandy Garner. 2006. Problem distributions in a CS1 course. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. 165–173.

[46] David W Scott. 1979. On optimal and data-based histograms. *Biometrika* 66, 3 (1979), 605–610.

[47] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 15–26.

[48] Teemu Sirkiä. 2012. Recognizing programming misconceptions-an analysis of the data collected from the uuhistle program simulation tool. (2012).

[49] Derek Sleeman, Ralph T Putnam, Juliet Baxter, and Laiani Kuspa. 1986. Pascal and high school students: A study of errors. *Journal of Educational Computing Research* 2, 1 (1986), 5–23.

[50] Juha Sorva et al. 2012. *Visual program simulation in introductory programming education*. Aalto University.

[51] Yuta Taniguchi, Tsubasa Minematsu, Fumiya Okubo, and Atsushi Shimada. 2022. Visualizing Source-Code Evolution for Understanding Class-Wide Programming Processes. *Sustainability* 14, 13 (2022), 8084.

[52] April Yi Wang, Yan Chen, John Joon Young Chung, Christopher Brooks, and Steve Oney. 2021. PuzzleMe: Leveraging Peer Assessment for In-Class Programming Exercises. *Proceedings of the ACM on Human-Computer Interaction* 5, CSCW2 (2021), 1–24.

[53] Mike Wu, Noah Goodman, Chris Piech, and Chelsea Finn. 2021. Prototransformer: A meta-learning approach to providing student feedback. *arXiv preprint arXiv:2107.14035* (2021).

[54] Ashley Ge Zhang, Yan Chen, and Steve Oney. 2023. RunEx: Augmenting Regular-Expression Code Search with Runtime Values. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 139–147.

[55] Ashley Ge Zhang, Yan Chen, and Steve Oney. 2023. Vizprog: Identifying misunderstandings by visualizing students' coding progress. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–16.

[56] Xin Zhou, DongGyun Han, and David Lo. 2021. Assessing generalizability of codebert. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 425–436.

## A MISTAKES PARTICIPANTS FOUND IN OPEN-ENDED QUESTIONS

## B POST-STUDY SURVEY QUESTIONS

## C BASELINE SYSTEM

Figure 5 shows the baseline system's user interface, which is derived from RunEx and incorporating OverCode's clustering results. Each code block (Figure 5g) represents either a cluster of correct solutions, or a single incorrect solutions, as computed by OverCode. To maintain a fair comparison, we take the exact same prototype from RunEx [54], except for that here each code block represents a cluster of solutions, while in the original prototype each code block represents a single solution. In the baseline system's user interface, code blocks are displayed in a grid view. In addition, users can easily create search queries in the baseline system by simply selecting code lines and typing in conditions to search through the entire solution set (Figure 5h). Each query is displayed alongside with descriptive statistics (Figure 5d). The systems allows for set operations on the queries (Figure 5e). The baseline system supports participants both view students' solutions in clusters and search through the whole solution set with text matching and runtime values. Users can easily filter the dataset through clicking the queries and the check box (Figure 5f). A thorough evaluation on the usability and effectiveness of the baseline system can be found in Zhang et al.'s paper [54].

## D PROMPT

We used the following prompt for the determination:

> '''
> Imagine you are a Python tutor, and there are some code samples for you to review. For each code sample below, there are some Attribute/Syntax/Value/Name/Type/Logical Errors. Read through it line by line and identify Errors based on the original Compile Error and the problem description given below. If you find out an Error, please do double check by reading codes around it. Be really careful while dealing with variables' names. Output all errors and keep the error description short. The output must strictly follow the Output Format below.
> Problem Description:
> {Problem Description}
>
> Output Format:
> ` ` `
> ERROR #: ERROR TYPE: ERROR Description | Line Number #: "Code"
>
> ERROR #: ERROR TYPE: ERROR Description | Line Number #: "Code"
> ` ` `
> Code Sample Example:
> ` ` `
> {Input Example}
> ` ` `
> Output Example:

| PID | Condition | Mistakes | Count |
|---|---|---|---|
| P1 | Baseline | None | 0 |
| | CFlow | mix up variable names, looping over empty dictionary instead of the word, use variables that are not defined, initialize as 0 for the first time encountered ant not adding 1 , looping over the keywords of an empty dictionary" | 5 |
| P2 | Baseline | concatenate an append statement to a str variable, missing quotation marks around "d" and "ed", use wrong conditional statement for modifying the word" | 3 |
| | CFlow | mix up variable name, syntax error in writing conditions, not referring to the "counts" variable correctly, incorrect use of the "str.split() method" | 4 |
| P3 | Baseline | indentation error | 1 |
| | CFlow | syntax error in writing condition, use variables that are not defined, using "is" instead of "==" when checking str values, use "()" to index dictionary instead of "[]" | 4 |
| P4 | Baseline | not define past_tense, syntax error on adding "d" or "ed" | 2 |
| | CFlow | looping over empty dictionary instead of the word, syntax error in writing conditions, use variables that are not defined, Key Error, indentation error, type error, attribute error, using str.split() when it is not necessary, mix up variable names, missing colon in for loop" | 10 |
| P5 | Baseline | Name Error, Key Error, Logical Error, type error, syntax error in wriitng conditions, loop over wrong data type | 6 |
| | CFlow | use append method on a str variable, incorrectly format append, modify the word but not assign it to any variable, concatenate an append statement to a str variable | 4 |
| P6 | Baseline | Name Error, Key Error, Logical Error, Syntax Error | 4 |
| | CFlow | modify the word but not assign it to any variable, add quotation marks around variable names, append "d" to the list instead of the modified word, add "d" to word[-1] instead of word, use "==" instead "=" to assign values, trying to reassign the append statement to the list, use wrong conditional statement for modifying the word | 7 |
| P7 | Baseline | mix up variable names, missing loops, use variables that are not defined | 3 |
| | CFlow | "append "d" to the list instead of the modified word, attribute error, syntax error, modify the word but not assign it to any variable, modify the word but not append it to the list, use wrong index when checking the last letter of word" | 6 |
| P8 | Baseline | use append method on a str variable, modify the word but not assign it to any variable, use wrong index when checking the last letter of word" | 3 |
| | CFlow | syntax error in writing conditions, do not understand how loop over str works | 2 |
| P9 | Baseline | None | 0 |
| | CFlow | indentation error, missing content after for loop | 2 |
| P10 | Baseline | None | 0 |
| | CFlow | modify dictionary values without initializing it, mix up variable names, looping over empty dictionary instead of the word, use variables that are not defined | 4 |
| P11 | Baseline | Name Error, Key Error, incorrectly index letters in str variable | 3 |
| | CFlow | incorrectly format append, compare a str variable with a list using "<", overwriting the list with a word instead of updating the variable, updating the index instead of the word, initialize list with a str variable | 5 |
| P12 | Baseline | check the whole word instead of the letter in conditional statements, use variables that are not defined, mix up variable names | 3 |
| | CFlow | append "d" to the list instead of the modified word, use append method on a str variable, use variable undefined, use wrong conditional statement for modifying the word, use "=" instead of "==" when checking the end of word, indentation error | 6 |
| P13 | Baseline | modify the word but not append it to the list | 1 |
| | CFlow | mix up variable names, indentation error, modify dictionary values without initializing it, check if letter is in the original str variable instead of the new dictionary | 4 |
| P14 | Baseline | missing conditional statement, Logical Error, wrong conditional statement for initializing dictionary | 3 |
| | CFlow | append "d" to the list instead of the modified word, modify the word but not assign it to any variable, modify the original list "word" but not creating a new list "past_tense" | 3 |
| P15 | Baseline | mix up variable names, check if letter is in the original str variable instead of the new dictionary, indentation error" | 3 |
| | CFlow | use append method on a str variable, use the "join" method instead of "append", incorrectly format append, trying to reassign the append statement to the list, concatenate an append statement to a str variable, missing content inside the loop | 6 |
| P16 | Baseline | use variables that are not defined, wrong conditional statement for initializing dictionary, loop over wrong data type | 3 |
| | CFlow | not change the word before append it to list, use ""e" in word" to check the end of a word, indentation error, directly modify word without conditional statements, syntax error on adding "d" or "ed" | 5 |

**Table 1: Mistakes participants found in open-ended questions**

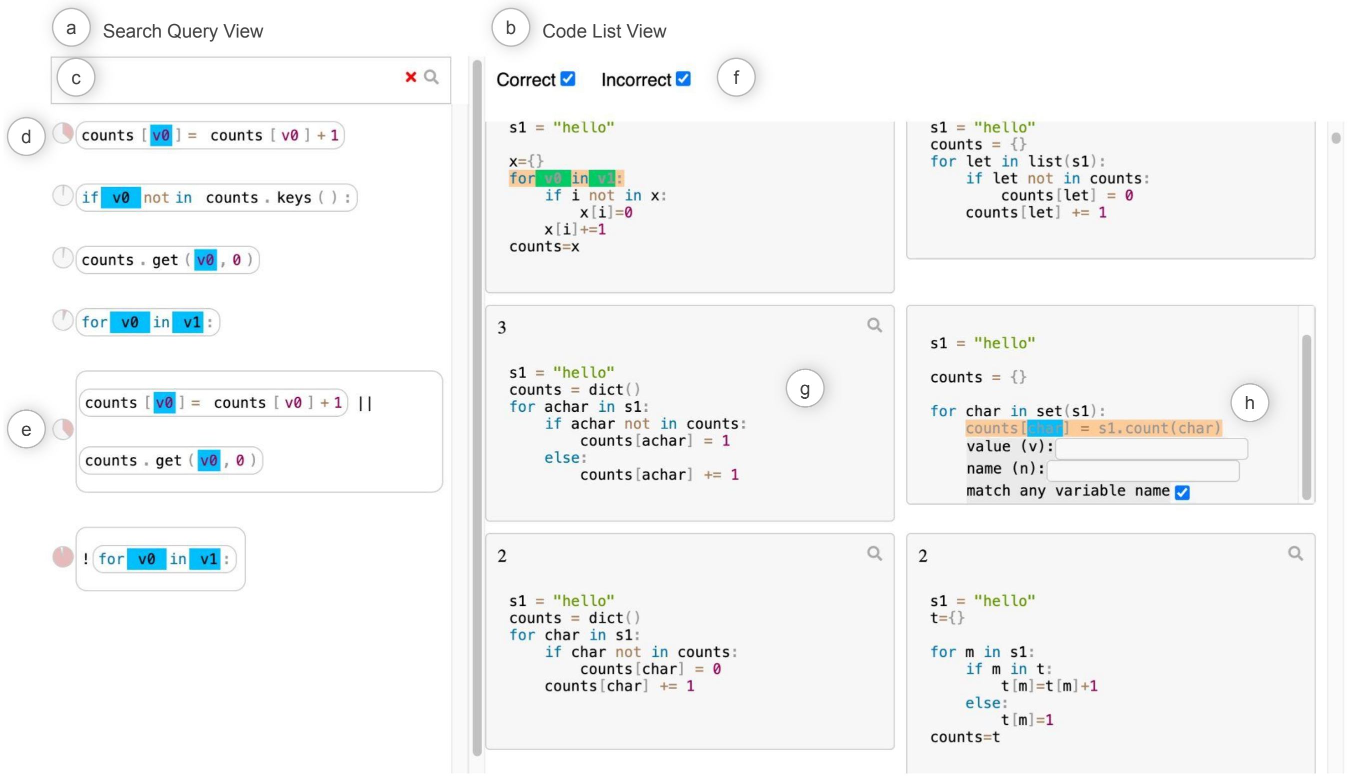
| Statement | Mean | SD | Median | Iqr |
|---|---|---|---|---|
| S1: Compared to RunEx, CFlow takes less time to identify students' misconception. | 6.06 | 1.48 | 7.00 | 1.00 |
| S2: Compared to RunEx, CFlow is less helpful in understanding students' approaches. | 3.38 | 1.80 | 2.50 | 3.00 |
| S3: Compared to RunEx, CFlow better supported me to explore students' solutions. | 4.63 | 1.62 | 5.00 | 3.00 |
| S4: Compared to RunEx, CFlow is less helpful in understanding the code structure information. | 2.94 | 1.48 | 3.00 | 2.00 |
| S5: Compared to RunEx, I would prefer to use CFlow for similar tasks in the future. | 5.63 | 1.73 | 6.00 | 2.00 |

**Table 2: Post-study survey results of comparing CFlow and the baseline system. Participants were asked to rate their agreement with various statements on a scale from 1 to 7, where 1 indicated complete disagreement. To ensure impartiality, we replaced the term Baseline with RunEx to prevent participants from discerning which system was the baseline.**

~~~~
```
{Output Example}
```
(Line Number starts from 0. Stop when there is no
error left. No more than 7 ERRORS!! Make sure that
there is no repeated ERROR!!)

Think step by step, and do self-reflection and self-
correction before output your answer! Do double
check across the code lines if you think you found a
variable that is not defined!!!

Original Compile Error : {OEmessage}
'''
~~~~



```
counts [ v0 ] = counts [ v0 ] + 1

if v0 not in counts . keys ( ) :

counts . get ( v0 , 0 )

for v0 in v1 :

counts [ v0 ] = counts [ v0 ] + 1 ||

counts . get ( v0 , 0 )

! for v0 in v1 :
```

```
s1 = "hello"

x={}
for v0 in v1:
    if i not in x:
        x[i]=0
    x[i]+=1
counts=x
```

```
s1 = "hello"
counts = {}
for let in list(s1):
    if let not in counts:
        counts[let] = 0
    counts[let] += 1
```

```
3

s1 = "hello"
counts = dict()
for achar in s1:
    if achar not in counts:
        counts[achar] = 1
    else:
        counts[achar] += 1
```

```
s1 = "hello"

counts = {}

for char in set(s1):
    counts[char] = s1.count(char)
    value (v):
    name (n):
    match any variable name ✓
```

```
2

s1 = "hello"
counts = dict()
for char in s1:
    if char not in counts:
        counts[char] = 0
    counts[char] += 1
```

```
2

s1 = "hello"
t={}

for m in s1:
    if m in t:
        t[m]=t[m]+1
    else:
        t[m]=1
counts=t
```

**Figure 5: The baseline system's user interface, derived from RunEx and incorporating OverCode's clustering results, features two main views: the Search Query View (a) and the Code List View (b). In the Code List View, each code block (g) represents a cluster of solutions with identical computation results, with a number in the upper left corner indicating the cluster size. Users can search for specific code patterns using runtime values and text matching (h), with each query displayed in the Search Query View alongside descriptive statistics (d). Queries can be entered directly into the search bar (c). Additionally, the system allows for set operations on these queries (e). Users can filter the code blocks in the Code List View by clicking on a query or using the checkbox (f).**