

Code Patternz - A tool to record the programming process

*Renaldo Williams
Dan Garcia, Ed.*



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-213

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-213.html>

August 31, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would like to thank my advisor Dan Garcia, who gave me the freedom to pursue an area that I found interesting in Computer Science Education and kept me motivated.

I would like to thank Professor John DeNero for providing guidance on the technical project. I would like to thank Professor Armando Fox, Audrey Sillers, Susanne Kauer, Shirley Salanio, Michael Sun, and others for their support throughout my years at UC Berkeley. Since I had to start working full-time, I thank my manager, Mark Takacs, for being supportive.

Finally, and most importantly, I would like to thank my parents, Verna and Alty Williams, my sisters, Renee and Andrea, my close friends, Sean, Josia, and MJ, and my partner, Candace, for being there for me when times were tough.

Code Patternz

A tool to record the programming process

Code Patternz - A tool to record the programming process

by Renaldo Williams

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee:



Professor Daniel D. Garcia
Research Advisor

8/20/2021

(Date)

* * * * *



Professor John DeNero
Second Reader

8/20/2021

(Date)

1. Introduction	5
2. Background	6
2.1. Code editors	6
2.2. Web-based code editors at UC Berkeley	7
3. Related Work	8
4. Overview of the Study	10
5. Design and Implementation of Code Patternz	14
5.1. Planning	14
5.1.1. Goal Setting	14
5.1.2. Testing Code Editors	16
5.1.3. Human Subjects Testing	17
5.2. Interface Design	18
5.2.1. User Interface and Features	18
5.2.2. Usability experiments	19
5.2.3. Final User Interface Flow	21
5.3. Architecture	30
5.3.1. Back-End Server	31
5.3.2. Database	33
5.3.3. Docker	34
5.3.4. Front-End	36
6. Data Analysis	41
6.1. Exploratory Data Analysis	41
6.2. Student Code Patterns	46
7. Future Work	52
8. Conclusion	53
9. Follow your inner voice	54
10. Appendix	55
10.1. Consent Form Human Subjects	55
10.2. Programming Problems	56
10.3. Sample of Survey Results	57
11. References	60

Abstract

“Life is a journey, not a destination.”
— *Ralph Waldo Emerson*

Many programming courses at Universities follow the typical pattern: (1) students are given programming assignments, (2) students go through the process of figuring out what to write to solve the programming assignment, (3) students write their solution, and finally (4) students submit their code to be graded. Within these processes, the *second* and *third* process is of interest to us as they focus on *what* students write when creating their solution. However, before we can focus on what students write, we must find a way to *capture* the details of what code is written in a way that is not disruptive to the student.

In this paper, we describe **Code Patternz**, a tool created to record the programming process, so that we can find patterns in the steps that students take while writing code. We are interested not so much in the correctness of their solution, but in the *coding path* that students follow when solving the problems. Typically, a student's final submission does not provide insight into this type of valuable information because the *process* (e.g., top-down, bottom-up, tests-first, etc.) and intermediate attempts are not stored.

To capture this information, we built a web-based code editor, and included three Python programming problems that students in an introductory computer science course UC Berkeley had to solve. While solving the problems, the code-editor tracked the sequence of characters the students typed and stored these characters in a database for later analysis. By collecting this granular information, it opens up the possibility to learn from the patterns we might discover across students during the code writing process.

One of the challenges with such an experiment was building a system that could manage the data load of capturing *every character students typed* when a large number of students were using the system at the same time. In this paper, we discuss the architecture of the Code Patternz system, and some of the tradeoffs we had to make. The tool has two sections; the first presents questions and captures the code written. The second is the *analysis* section used to “replay” the code as if it were being written live. It also exports the stored data in a form that makes it easy to perform later analysis. We provide summary statistics as well as a “deep dive” into a few of the outlier student traces for the data collected from the UC Berkeley students.

There is a great amount of information that can be extracted through the analysis of the sequence of characters written by students when solving programming problems. The patterns we discover can help guide further instruction; watching *where* students struggle can inform future curriculum and intervention efforts. We hope our tool can provide a platform to ask these questions easily.

1. Introduction

In modern day public Universities, large introductory programming courses are the norm, fueled by ample employment opportunities available in industry. According to the U.S. Department of Labor, Bureau of Labor Statistics, the employment opportunity for Computer Science (CS) and Information Technology (IT) occupations is expected to grow 12% from 2018 - 2028 [2]. These fields are projected to grow faster than the average of all other occupations[3], so the increase in student enrollment in computer science and other programming courses is a logical occurrence.

Undergraduate Computer Science students are required to complete a series of programming courses that involve a large amount of programming. It is not uncommon to see class enrollment well over 1000, such as UC Berkeley's CS61A course, which taught over 1700 students in the Fall of 2021. It is no surprise that the number of students majoring in Computer Science has more than doubled at many institutions from 2013 to 2017 [4]. With such large class sizes, an interesting question is: what are all the ways these thousands of students are approaching their programming assignments, and could we record and analyze them to discover interesting patterns?

Whether computer programming is declared “hard” or “easy” to learn is a matter of some debate (Luxton-Reilly [20]). What *is* clear is that beginning students take very different paths when given a problem. Some make a bee-line to the solution (by intention or luck!), some have many failed attempts and eventually get their code to work, and some unfortunately never get there. Much of the existing research on Computer Science education focuses on interesting ways to *teach* programming [1], with less attention given to studying the *process*.

We believe in the motto that, “it is the journey, not the destination that matters”. Tools that capture coding sequences can provide a view into the journey that programmers travel when enroute to finding a solution. They provide a *microscope* into the mind of the programmer, indirectly translating their thoughts into actions. For example, if we see a sequence of characters written, then deleted, then tests run and failed multiple times (with lots of pauses in the middle) it may be indicative of someone who is unsure of how to proceed. We can't read the mind of the student, but we can tell that their path forward is unclear.

This report describes **Code Patternz**, a web-based code editor we built that logs every character a student presses while working on programming problems in the tool. One of the main goals was to make it easily accessible to any student. To accomplish this, we made the tool web-based, since a browser is all that is needed. We also understood that asking students to download and install key-logging software to their personal computers was a non-starter. One of the real risks of this research is the uncertainty of whether collecting large amounts of keystroke data would yield any interesting findings. That said, we approached the project with an optimistic mindset that “there may be gold in them thar hills (of data)”. . The Code Patternz tool for collecting the data is the first step in helping us discover patterns related to *how* students solve programming problems on a code sequence level.

2. Background

Modern Integrated Development Environments (IDE) and online code editors make it easier for software developers to write code through the support of rich features such as syntax highlighting, code completion, error reporting, and debugging information[18]. In this section, we provide an overview of the two main types of software development environments for writing code, and briefly discuss how UC Berkeley uses online code editors in its courses.

2.1. Code editors

In the 1980's, developers mainly used Emacs and vi (now usually Vim) to write code. Both don't include the intuitive graphical user interfaces (GUI) that modern IDE's and web-based code editors have, but still have a strong following. Most developers today, however, use more modern tools for software development. There are two main types of software development environments: locally downloaded, and web based interfaces. For example, UC Berkeley's CS160 "User Interface Design and Development" course, which teaches students about Android development, uses Android Studio -- an application that students download and use for their programming assignments. Other regularly downloaded Integrated Development Environments (IDE) include Eclipse, IntelliJ, and Visual Studio[24]. One of the main downsides of these types of development environments is the heterogeneity of operating systems, code editors, IDEs, compilers, interpreters, additional libraries that need to be installed to ensure the software works as expected[9]. Most of the time, the software installer takes care of installing dependencies, but sometimes not.

The second type of environment is web-based development environments that are based online, residing on a remote server and accessible via a web browser. The main advantage of a web-based code editor is that nothing has to be installed; the system resides on a server, which already has all the configuration in place to run the code. Another advantage of web-based development environments is its tight integration with Massively Open Online Courses (MOOCs)[9]. Since MOOCs are already accessed through a web browser, they work well with web-based development tools that are based on HTML and Javascript, because both curriculum and coding environment can be present on the same webpage[10].

In the past decade, several different web-based code editors have attracted a large fan-base, with the most well-known being CodePen[6]. One of the main components of web-based editors are the embeddable code editors that are based on Javascript. For example, Ace editor and CodeMirror offer rich features that resemble native IDE's.

Another benefit of web-based code editors is their "Google Docs"-like concurrent editing collaboration features. They allow developers to work in the same document as they simultaneously author code, convenient for educators trying to teach others certain programming concepts. Early web-based code editors could only run Javascript. However in recent years, more web-based code editors have been blossoming with features, and can now support other languages, such as Python or Java.

2.2. Web-based code editors at UC Berkeley

In general, web-based code editors have become more popular at Universities in the past decade because of the ease of access. All the student has to do is go to the website, login, start coding, and submit. For example, the CS100/200A “Principles and Techniques of Data Science” course at UC Berkeley uses Jupyter Notebooks to teach Python for data science to its 800+ students.

These online code editors for large CS courses present several benefits. The first is that it becomes easier to monitor the steps that a student takes to solve a coding problem, since keystrokes can be tracked using Javascript. Another benefit is that the code written by students can be stored in one central database, creating fewer barriers for code analysis.

The Code Patternz code editor records characters pressed in order to find key insights into the programming *process*. The tool does this by monitoring the interactions (i.e. keystrokes, cursor clicks) of students as each student attempts to solve a coding problem. The overall goal is to study the code patterns of students solving the same problem in order to understand similarities and differences between students, and learn where students are struggling (or succeeding), to influence curriculum and instruction.

3. Related Work

There have been several studies that developed software tools to record how students write code within a software development environment. Lyulina et al. create a plugin for IntelliJ-based IDEs (PyCharm, IntelliJ IDEA, CLion). The researchers capture snapshots of code and IDE interaction data with the intention to collect fine-grained data of how students solve their programming assignments and analyze their learning progress. The tool has three main components: (1) an IDE plugin to track the process of how students solve programming problems within the IDE; (2) TaskTracker-server to gather data remotely, collect solutions of multiple students in one place and; (3) Data post-processing tool for basic processing of data collected by TaskTracker-plugin. The researchers clearly show how a software tool can be used to capture granular student data and the same tool can be used for analysis.

Piech et al. also instrumented an Integrated Development Environment (IDE) named Eclipse in order to take snapshots whenever the student would compile their code. The researchers used various statistical methods such as machine learning algorithms and Hidden Markov models which allowed them to create a dissimilarity matrix for clustering of 2000 snapshots to find patterns and clusters in the snapshots. The researchers point out that using these techniques are advantageous because manually capturing granular student data of keystrokes can cause privacy concerns and it can be tedious to interpret raw snapshots of code .

For finding patterns in error messages, Marceau et al. created a programming environment named DrRocket, which uses human-factors research to explore the effectiveness of error messages. This was in response to the research efforts to make compiler errors more suitable for teaching by writing error messages that are accompanied by hints and explanations.

Robinson and Carroll implemented an open-source online learning platform to aid in teaching and assessment of computer programming in large classes. They discuss the implementation of the tool and how it addresses the pedagogical concerns of both the learning environment and assessment of programming. Staubitz et al. similarly offers a list of helpful tools for automated assessment of programming exercises that work well with MOOC components. Results from an associated survey showed that professionals learning programming were less interested in theoretical content and more in practical experience.

Mutiawani and Juwita try to tackle the difficulties that many undergraduate students face when trying to comprehend logic and programming by focusing on e-learning systems. An alternative e-learning solution is introduced that includes interactive multimedia to help undergraduate students learn introductory programming. One of the benefits is that students only need a web-browser to open the application on a computer or tablet which is connected to the internet.

For general tools for assistance, Koorsse et al. discuss programming assistance tools (PATs) that are specifically designed to teach novice students how to program. PATs support several features including problem solving, algorithm design, assist with learning syntax for a particular programming language, and partial compiling to quickly check output and operation of a code block. PATs can also make use of visualization and animation techniques because programming concepts, data structures and algorithms are abstract. They argue that visualization techniques can be used to help novice programmers develop more effectively.

And finally, though there is not much research logging actions, Meier et al. used log files using a programming environment named *Thorny* to capture student behavior in MOOCs. They were interested in observing behavioral patterns in different learners, and how to distinguish patterns in problem solving.

All of these works aim to create or modify existing software to help students improve their programming problem process and find patterns in that process.

4. Overview of the Study

The purpose of the study was to build a tool that could help the CS education research community gain a better understanding of the different ways that students solve programming problems on a granular level.

The study was conducted during the summer of 2021 with students from a UC Berkeley course, CS61A. CS61A is an introductory programming course that teaches various Computer Science concepts using the Python programming language. The course typically has one of the largest enrollments for an introductory programming class in the country; in the Fall 2021 the course had over 1700 students! The course enrollment during the summer when this study was held was roughly 490 students.

In order to participate in the study, the students attempted to solve three programming problems as the extra credit portion for one of their weekly labs. In order to capture the code that the students authored to solve the problems, we created a web application with an embedded code editor. The tool presented the programming problems and the editor in a single window so that students could see a problem while simultaneously seeing the code they would write to solve the problem. The programming language that the students used to solve the problems was Python, the language taught in CS61A. The students were given a description of the problem and skeleton code that would be used as initial code to start the problem.

Below is an example of the first programming problem that the students had to solve.

Problem Background Topics

Tuple Reverse View Instructions

Write the reverse procedure which operates on tuples. The function takes an input tuple, seq, and returns a tuple with the same items in reversed order. It does not reverse any items in the tuple and does not modify the original tuple.

For example, if:

```
x = (1, 2, 3, 4, 5)
```

then:

```
reverse(x)
```

should return:

```
(5, 4, 3, 2, 1)
```

Another example is, if:

```
y = (1, 2, (3, 4), 5)
```

then:

```
reverse(y)
```

should return:

```
(5, (3, 4), 2, 1)
```

For more information on [tuples](#), see the Tab above named [Background Topics](#).

Then, write your code in the editor on the left side and click [Run Tests](#) to test if your code passes the tests. Click [Submit Code](#) when you are done (i.e. have gotten as far as you can). You can submit as many times as you like. Click [Next Problem](#) to go to the next problem.

After you've done all the problems, click the [Home Button](#) to go back to the [Question Bank](#) and click [Open Survey](#). Fill out the short survey and Click [Submit](#). And your all done!

Remember to type your code directly in the [code editor](#) on the left so that your code pattern can be analyzed. Your code pattern may be shared with other students however your code will be anonymous, meaning no one will know who wrote this code. You are obviously free to share this code with whomever you like since its your code.

Figure 1. The first programming problem that the students had to solve for the study.

The first question ([Figure 1](#)) served as an initial question, designed to be the easiest of the three problems. The question asked students to reverse a tuple, which could be done in one line of code if the students knew the proper syntax for list slicing in Python. Since many of the students were new to programming, we did not expect them to know slicing unless they already learned it in the course. This question also served as a “softball” question to help students become more acquainted with the user interface. Below is an example of the starter code for this problem.

```

def reverse(seq):
    """Takes an input tuple, seq, and returns a tuple with the same items in
    reversed order. Does not reverse any items in the tuple and does not
    modify the original tuple.

    Arguments:
    seq -- The tuple for which we return a tuple with the items reversed.

    >>> x = (1, 2, 3, 4, 5)
    >>> reverse(x)
    (5, 4, 3, 2, 1)
    >>> x
    (1, 2, 3, 4, 5)
    >>> y = (1, 2, (3, 4), 5)
    >>> reverse(y)
    (5, (3, 4), 2, 1)
    """
    #***DON'T MODIFY THE CODE ABOVE***

    """ YOUR CODE HERE. """

```

Code Snippet 1: The skeleton code for problem number one

The starter code included the name of the function that was going to be tested, and the doc test that was used to test whether the students' code returned the value that matched that of the doc test. One of our concerns was that since the doc test was visible, students could "game" the system by creating *if* statements that return the same values as the doc tests, and hence pass all the tests. That is *not* how we want students to pass the tests. We want students to solve the questions by adhering to the problem statement and writing code that works with any input, not just the inputs tested by the doc test.

While the students typed their code for the programming problems (Code Snippet 1), the code editor would log the keys typed. The code they wrote is what we use as data to understand their coding path enroute to their solution. We emphasized to students that if they did not solve the questions correctly, that was okay. It was more important that the students not feel pressured to Google (or ask their friends) the correct answer, but instead feel comfortable to attempt the question without outside help. This allowed the data to be representative of each student's full capability. After getting to a point where the student felt they had gotten as far as they could for each problem, the student would *submit* their code by clicking the submit button on the user interface.

Upon completion of a programming problems, the students completed a short 5-minute online survey which asked the following questions:

1. For the problems you were able to solve (i.e., all tests passed), what approach did you take to find the solution? Help us understand what was going through your mind while coding.

2. For the problems you were not able to solve (i.e., not all the tests passed), what do you feel were barriers that prevented you from finding the solution? What approaches did you try?
3. Did you have any issues while using this application? Is there anything we could do to make it easier to use?

In addition, we mentioned that a small subset of students may be chosen for a follow up Zoom interview if we thought that their code patterns were interesting enough to warrant it. We also were clear to point out that it would take us time to do the data analysis, and the class would be over by then, so there would be no direct benefit to the students in the course for summer 2021. However, we informed them that the results from the study may benefit *future* CS61A students and help researchers at UC Berkeley gain a better understanding of students' programming process.

5. Design and Implementation of Code Patternz

In this section, we describe the planning, interface design, architecture, deployment of the Code Patternz tool for capturing the programming process. We also describe a basic data analysis engine we used to process the data produced by Code Patternz.

5.1. Planning

In the planning step, we created an outline of time allocation and goals of developing the software. In the beginning, it was difficult to create an exact description of how the end product would look because we did not know what features Code Patternz would need to have. So we focused mainly on the types of questions we wanted to ask and becoming comfortable with how we could translate the native coding experience to a web-based editor.

5.1.1. Goal Setting

The initial focus for us was to get a sense of what the code editor would need to support in terms of features to allow CS61A students to solve the programming problems in an online code editor. We met with summer 2021 instructors and discussed the concept of the study. The purpose of the meeting was first, get approval that the instructors were okay with using their students for the study, and second, to understand the type of programming environment that students in their course are accustomed to. We learned that typically the student coded in a native text editor to write their Python code, and used Terminal to run the file that contained the Python code. We endeavored to replicate that experience in the web based code editor.

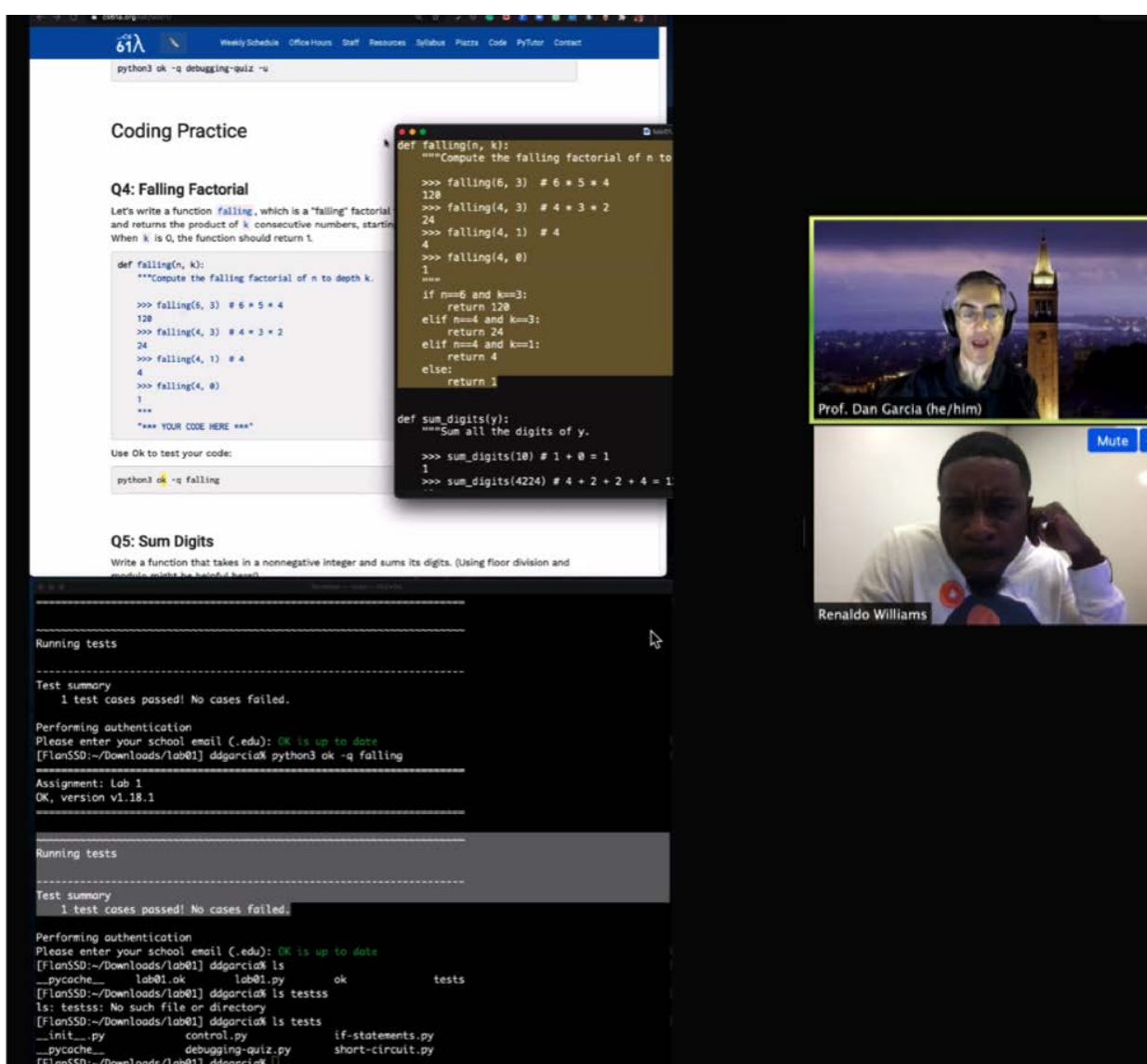


Figure 2: Testing the programming environment that students in CS61A were accustomed to.

Figure 2 shows a screenshot of a virtual meeting where the researchers solved programming problems in the same environment that students in the course used. The goal was to understand how to transfer this environment to an online experience without losing any of the performance and stability of the native development tools. We realized early on that we would have to transfer *everything* about the experience into the web: the display of the question (easy, just simple HTML), the editor to record their keystrokes (more difficult, we describe in section 5.3), and some system *to run* their python code. Implementing this last task proved to be among the most difficult technical challenges we had to overcome.

Some very basic initial steps we outlined that we needed to take to meet our goals were:

- Work with the CS61A instructors to understand how this study can be conducted in their course and when did it make sense to carry out the study
- Understand the type of development environment the student in CS61A would be used to.
- Figure out a way to run a browser-based terminal, since the students used the terminal a lot in the course to run their code.
- Figure out how a web-based terminal would connect to a remote server, since the terminal needed a computer to connect to in order to operate correctly.
- Figure out how the web-based terminal could run Python code
- Figure out what type of questions we would like to ask the students
- Figure out if it would be possible to replay the transcript of their code as if it were being typed live

5.1.2. Testing Code Editors

Many of our initial meetings involved testing online code editors to understand how they worked and noting down any interesting features we would want in our tool. Online code editors have many benefits which include zero setup, easy collaboration, and little cost to operate [21]. Some of the popular online code editors include CodePen, Code Sandbox, JS Fiddle, Coderpad, Leetcode, W3Schools, etc [21].

From observing other online code editors, we realized that from an architectural standpoint, we would need the typical web development stack, which consisted of a front-end web application, a web server to host the back-end logic, and a SQL database to store the data. We also needed to understand how the server would run Python code, since popular online code editors do not document how their back-end logic performs this task. We would have to figure it out ourselves.

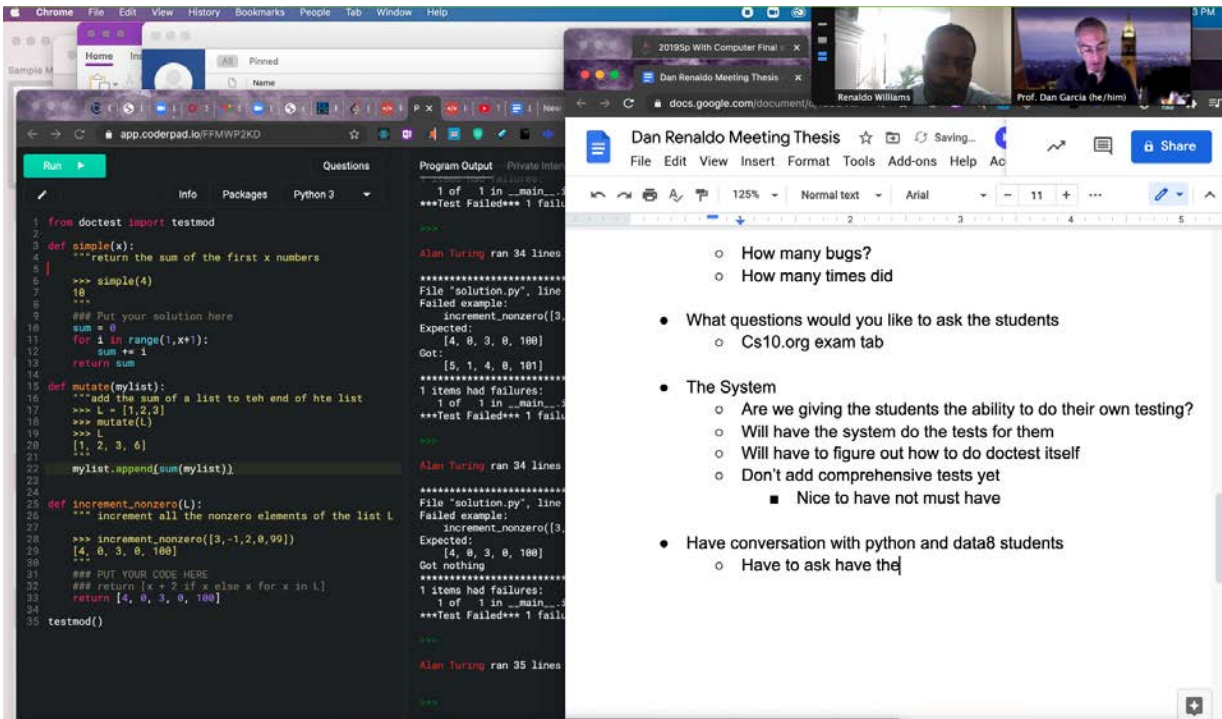


Figure 3: The researchers brainstorming sessions for how the online code editors work.

5.1.3. Human Subjects Testing

The web based code editor we developed was planned to be used by students in UC Berkeley's CS61A course during the summer of 2021. In order to perform the experiment on human subjects, an application had to be submitted to the Committee for Protection of Human Subjects (CPHS), UC Berkeley's Institutional Review Board (IRB). The application had to include key information about the study, including the Consent forms that students would need to agree to if they were to participate in the study. More details can be found in the Appendix. Section 10.1 illustrates the sample consent form. The approval process took longer than expected (6 - 8 weeks), because many details had to be hammered out, as the study involved human subjects and collecting personal information (students email). The development of the software occurred simultaneously with the human subjects approval process. Once our application was approved, we could perform the study on students once the student agreed to participate.

5.2. Interface Design

The interface design was an important aspect of this study. Not only did we want the interface to be intuitive but the interface also had to have a modern look and feel to it, especially since it would be used by young students in their freshman and sophomore year of college. In this section we describe three important factors - the requirements, usability experiments, and the final interface.

5.2.1. User Interface and Features

The user interface design was a crucial step in this research because the tool had to be easy to use since this experiment was carried out remotely. At the time of this study, most of the courses at UC Berkeley, including CS61A, had moved online due to COVID-19. This meant that the study could not be conducted in person, which would have been ideal, and the software would have to be intuitive to navigate around, in addition to having enough instructions to guide students through the experiment.

In order to understand what features the user interface had to have, we had to clarify what the software requirements should be. Some functional requirements we defined were:

- The system would allow students to login using their UC Berkeley email to ensure that participants were students at UC Berkeley.
- The system would provide students with a unique code (sent to the students' email) in order to sign in. Students would *not* use their Berkeley password to sign in.
- The system would need to present the consent form to students and would not let students continue until selecting if they agree or disagree to participate in the study.
- The system would show an initial overview video so that students would understand how to use the tool.
- The system would present three programming problems written as text that students would have to solve.
- The system would allow students to type their code directly in a code editor
- The system would allow students to test their code and receive output of whether the code passed or failed the tests.
- The system would allow students to submit their final code for each problem.
- The system would allow students to fill out a survey where they described their thought process for the problems.
- The system would allow students to replay the code that they wrote for each problem.

After specifying these requirements, we had a better idea of what exactly we had to build and test during our usability experiments.

5.2.2. Usability experiments

Usability testing refers to evaluating a product or service by testing it with representative users. Typically, during a test, participants will try to complete typical tasks while observers watch, listen and take notes. The goal is to identify any usability problems, collect qualitative and quantitative data and determine the participant's satisfaction with the product [16]. Usability testing was done with several people. Below in Figure 4 is an example of the usability testing session for Code Patternz.

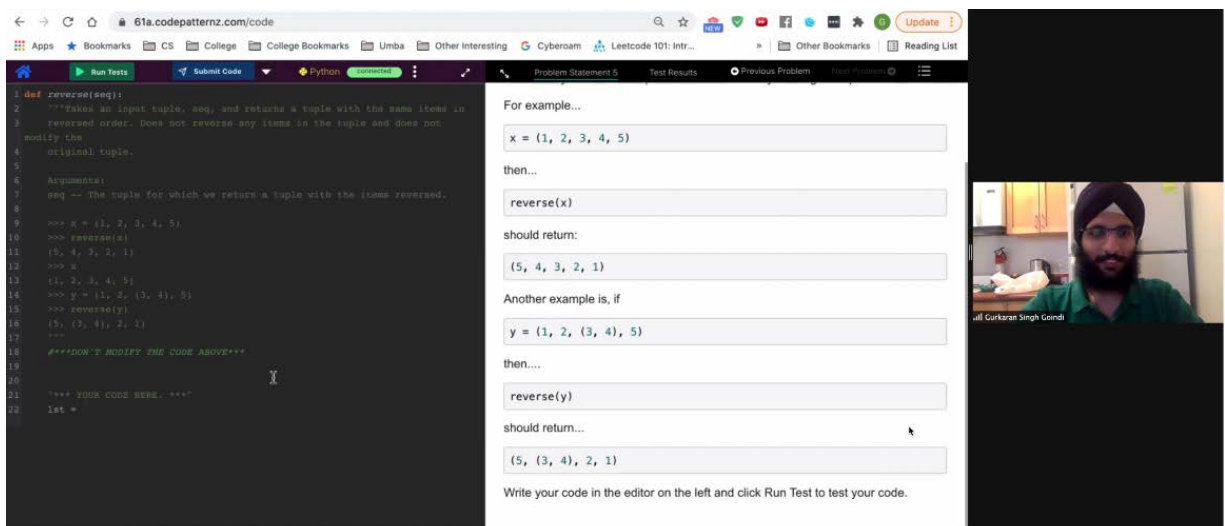


Figure 4: Usability testing with a TA for CS61A

Some key findings from the usability testing were:

- The Code Patternz code editor had to be very natural to use for coding. It should feel as natural as code editors used in the IDE's students are familiar with using. For example, we initially did not use a monospaced font, making it difficult to see if the code aligns -- an important aspect in Python. Figure 5 below shows the difference between a non-monospaced font and a monospaced font. Notice how easier it is to read the monospaced font because the letters are aligned vertically.

non-monospaced font

```
22 ptr = lst
23 result = 0
24
25 while ptr:
26     print(ptr.first**2)
27     result += ptr.first**2
28     ptr = ptr.rest
29
30 return result
```

monospaced font

```
22 ptr = lst
23 result = 0
24
25 while ptr:
26     print(ptr.first**2)
27     result += ptr.first**2
28     ptr = ptr.rest
29
30 return result
```

Figure 5: Non-monospaced font versus monospaced font.

- The editor needs to be able to run tests and show the result of the tests, otherwise students would not have an incentive to solve the problem. Figure 6 below shows the output of a failed test and a passed test. Notice the colors - the failed test has more red, and the passed test has more green. These colors served as a visual indicator of whether the students code output the correct values. We hoped that this test output would make the students see the code editor as a game that they had to pass and hence, not only have fun playing the game but also put in more effort to write code that passes the tests.

failed test

```
-----
RUNNING TESTS
-----
Doctests for square_linked_list
>>> lst1 = Link(1, Link(2, Link(3, Link(4)))
>>> square_linked_list(lst1)
100

# Test failed {}

# Expected
# 30
# but got
# 100

-----
Test Summary
0 test cases passed before encountering first failed test case

(End of Test Summary)
```

passed test

```
-----
RUNNING TESTS
-----
Test summary
1 test cases passed!

Good job! All test cases passed. Remember to Submit Code.

(End of Test Summary)
```

Figure 6: The output of a failed test versus the output of a passed test.

- The online code editor should not show too much information on the screen because all the information becomes too distracting to the eye. Since the code editor is situated in a web browser, efficient usage of the screen real estate was important. One of the initial mistakes we made was to make the editor have 3 panes for the code editor, the problem description, and an interactive terminal. After some usability testing, we realized three panes crowded the screen because our users found it difficult to read the problem description from the words being more condensed horizontally. The 2 pane layout was able to show the most important information while adding more horizontal space. This made the problem description easier to read, and the code editor easier to code in because each line could show more code horizontally.

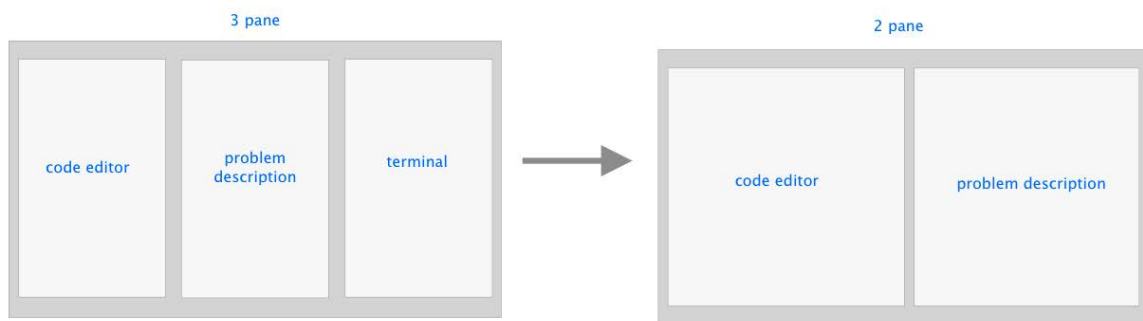


Figure 7: The 3 pane layout that was changed to a 2 pane layout.

5.2.3. Final User Interface Flow

Our goal was to make sure the final interface was well-designed, but also included creative illustrations so that the tool felt “fun” to use for students. Since the experiment was carried out in an introductory programming class consisting mostly of freshman and sophomore students, we wanted to make sure these young students enjoyed their programming experience.

Below is an explanation of each of the main pages of the Code Patternz web application. The url for the web application was <http://61a.codepatternz.com> but the website has been shut-down after the experiment concluded. We start with the landing page for Code Patternz in Figure 8, and after they log in are shown the main welcome screen in Figure 9. The page then automatically transitions to the consent form screen in Figure 10. After consenting (or not, in which case we don’t include them in our data analysis) and clicking “continue”, they are presented with the main welcome screen and a video introduction. On the left frame they now have an option to select the “Question Bank” displayed in Figure 12. Finally, Figure 13 shows

the instructions in a written format, Figure 14 displays the main code editor screen, Figure 15 contains the test result screen, and Figure 16 reveals the final test survey.

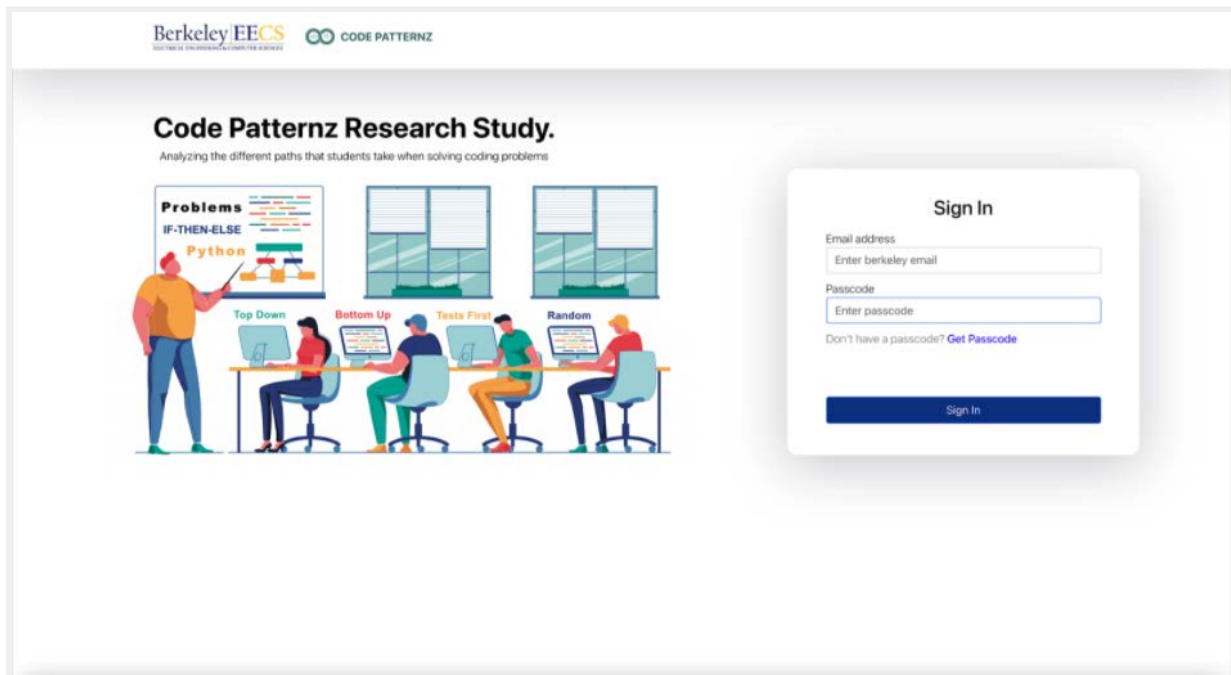


Figure 8: The initial landing page

The initial landing page was the first page that the students would see, so wanted the first impression to be creative and colorful. When first looking at the landing page, one thing that stands out is that it is colorful and the layout looks like a modern website. Another important distinction of the landing page is that the main image describes the purpose of the study. This would convey to students that we are interested in studying the different patterns students use to when approaching problems.

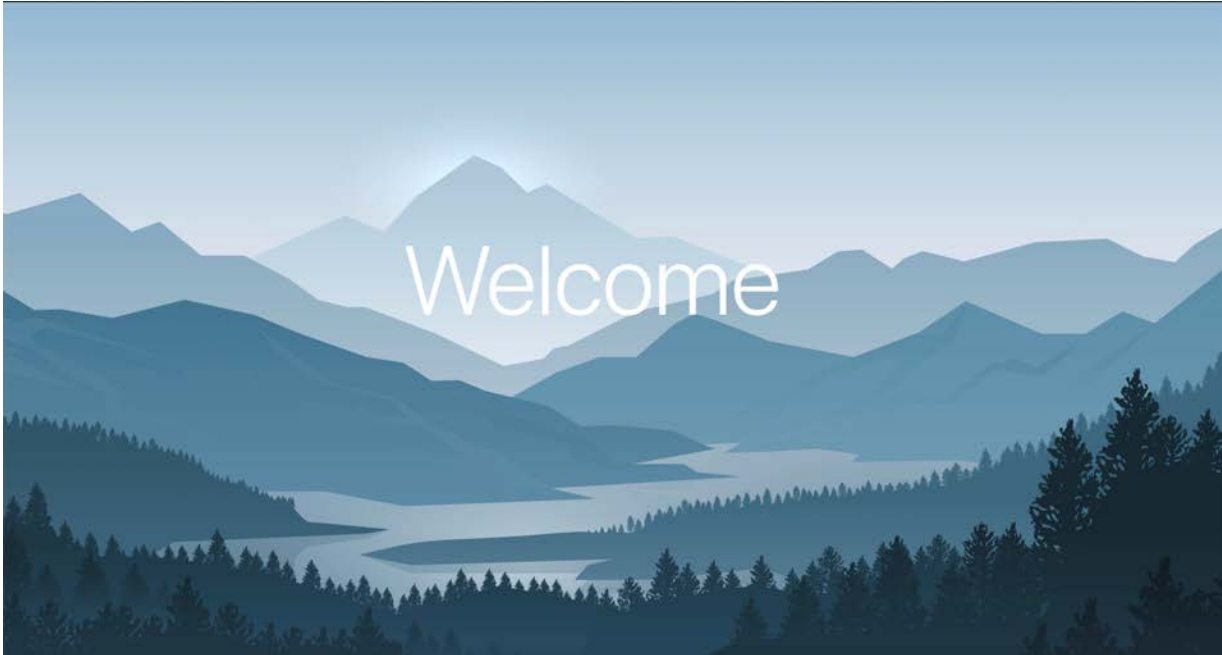



Figure 9: The welcome screen

We felt the welcome screen was a nice way to help students relax before entering into the tool. We did not want students to feel stressed or that participating in the research would be a burden.


Berkeley EECS  CODE PATTERNZ renaldo@berkeley.edu | [Sign Out](#)

Before you begin...

I have read the consent form and agree to participate.
 I have read the consent form and do not agree to participate.

[CONTINUE](#)

[18 or older, Download this version](#)
[Younger than 18, Download this version](#)

UNIVERSITY OF CALIFORNIA AT BERKELEY 
BERKELEY • DAVIS • IRVINE • LOS ANGELES • MERCED • RIVERSIDE • SAN DIEGO SAN FRANCISCO • SANTA BARBARA • SANTA CRUZ

CONSENT TO PARTICIPATE IN RESEARCH
Analysis of Code Patterns in CS61A

Key Information

- You are being invited to participate in a research study. Participation in research is completely voluntary.
- The purpose of the study is to gain a better understanding of the different ways that students solve programming problems in 61A and to understand common patterns in student code.

Figure 10: The consent form page

One necessary requirement of participating in the study was that students had to agree to sign a consent form. Since the class was held remotely, we embedded the consent form in the web application where the students would opt into participating in the study. One important clarification that we had to make was that students could decline to participate in the study, but still complete the three programming problems. The only difference is that we would exclude their data when performing the data analysis.

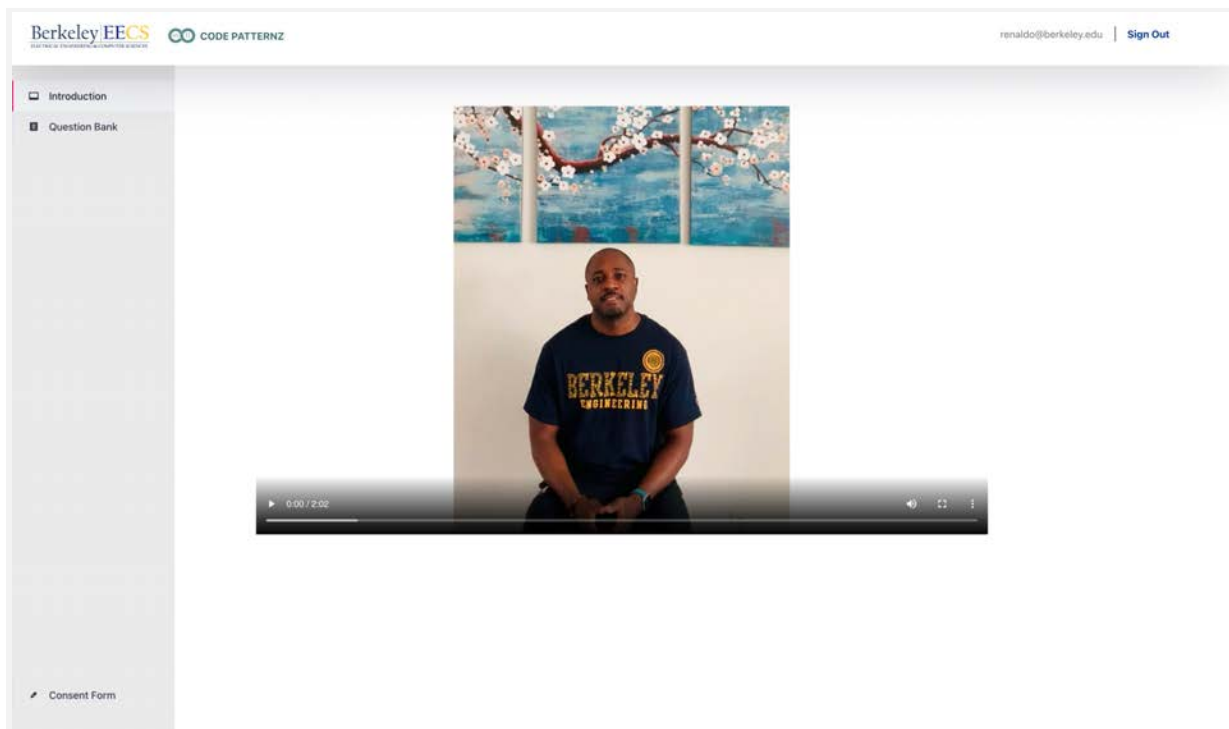


Figure 11: The video welcome screen

Immediately after the students made a decision with the consent form, we provided a welcome video from the author. It served as a way to make the tool more personable and to help the students connect more with one of the people conducting the study. The video also had a walk-through section that did an overview of the main features of the application, and demonstrated how to navigate the interface to do the programming problems.

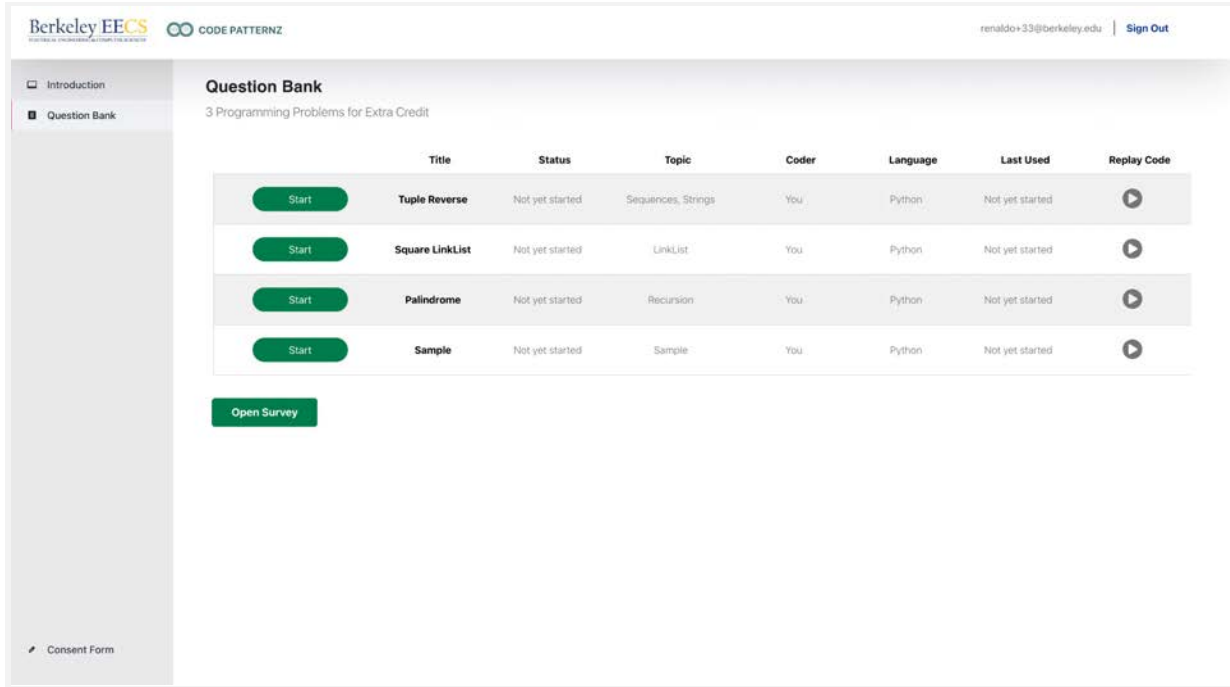


Figure 12: The question bank screen

The question bank is the final screen before students start working on the programming problems. The page was designed to provide important information about the problem that the student would solve, the status of whether the question had been started yet, and other important information (such as whether the student had submitted the problem). Notice that the *Replay Code* play button is grayed out because that feature was not implemented by the time the experiment was conducted. Based on the outcome of the study, students felt this screen was intuitive enough to know what to do.

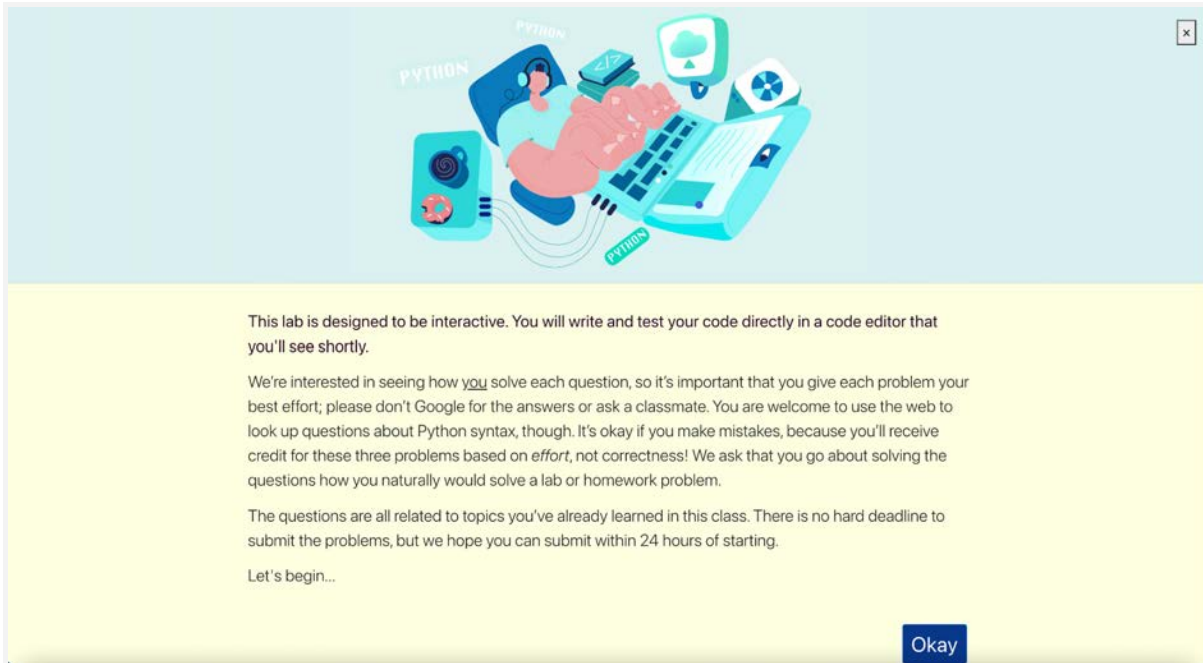


Figure 13: The instruction screen

The instructions screen automatically appears after the user clicks the Start button for the question. The instructions page served as a written reminder to some of the important information that was said in the introductory video on Figure 11. We wanted to emphasize that the problems were not based on correctness, but on effort. Our hope was that this would discourage students from Googling the correct answer.

```

1 def reverse(seq):
2     """Takes an input tuple, seq, and returns a tuple with the same items in
3     reversed order. Does not reverse any items in the tuple and does not modify the
4     original tuple.
5
6     Arguments:
7     seq -- The tuple for which we return a tuple with the items reversed.
8
9     >>> x = (1, 2, 3, 4, 5)
10    >>> reverse(x)
11    (5, 4, 3, 2, 1)
12    >>> x
13    (1, 2, 3, 4, 5)
14    >>> y = (1, 2, (3, 4), 5)
15    >>> reverse(y)
16    (5, (3, 4), 2, 1)
17    """
18    #***DON'T MODIFY THE CODE ABOVE***
19
20    #*** YOUR CODE HERE. ***
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38

```

Problem Statement 1 Test Results Previous Problem Next Problem

Problem Background Topics

Tuple Reverse

Write the reverse procedure which operates on tuples. The function takes an input tuple, `seq`, and returns a tuple with the same items in reversed order. It does not reverse any items in the tuple and does not modify the original tuple.

For example, if:

```
x = (1, 2, 3, 4, 5)
```

then:

```
reverse(x)
```

should return:

```
(5, 4, 3, 2, 1)
```

Another example is, if:

```
y = (1, 2, (3, 4), 5)
```

then:

```
reverse(y)
```

should return:

```
(5, (3, 4), 2, 1)
```

For more information on `tuples`, see the Tab above named `Background Topics`.

Then, write your code in the editor on the left side and click `Run Tests` to test if your code passes the tests. Click `Submit Code` when you are done (i.e. have gotten as far as you can). You can submit as many times as you like. Click `Next Problem` to go to the next problem.

After you've done all the problems, click the `Home Button` to go back to the `Question Bank` and click `Open`

Figure 14: The code editor (left) and problem statement (right)

After clicking the “Start” button from the “Question Bank” screen, students are presented with the screen shown in Figure 14, showing the code editor and problem statement. The code editor is the place where students would spend a majority of their time. Students read the problem description on the right, and write their code to solve the problem on the left. It was important that the problem description had proper font and spacing to make sure that the text was easy to read. Another important aspect of the right side is that there is a tab for Background Topics (not displayed by default). For example, this tab served as a way that students could read about a tuple data type.

The code editor had to be designed in such a way that it had some of the important features of native IDE’s, such as monospaced font, a run button, and a submit button. The code editor logs every character typed and every cursor movement and sends that information to the back-end database. Notice that in the top right portion of the page, there is a “Next Problem” button and a “Previous Problem” button (grayed out since we’re at the first problem). These buttons served as a convenient way that students could go on to other problems without having to go back to the question back screen (Figure 12). Based on the outcome of the study and feedback, students felt that the user experience for the problem statement and code editor were good. Some students mentioned that some of the content took a while to load; after we added load balancers discussed below, this helped with reducing that overhead time.

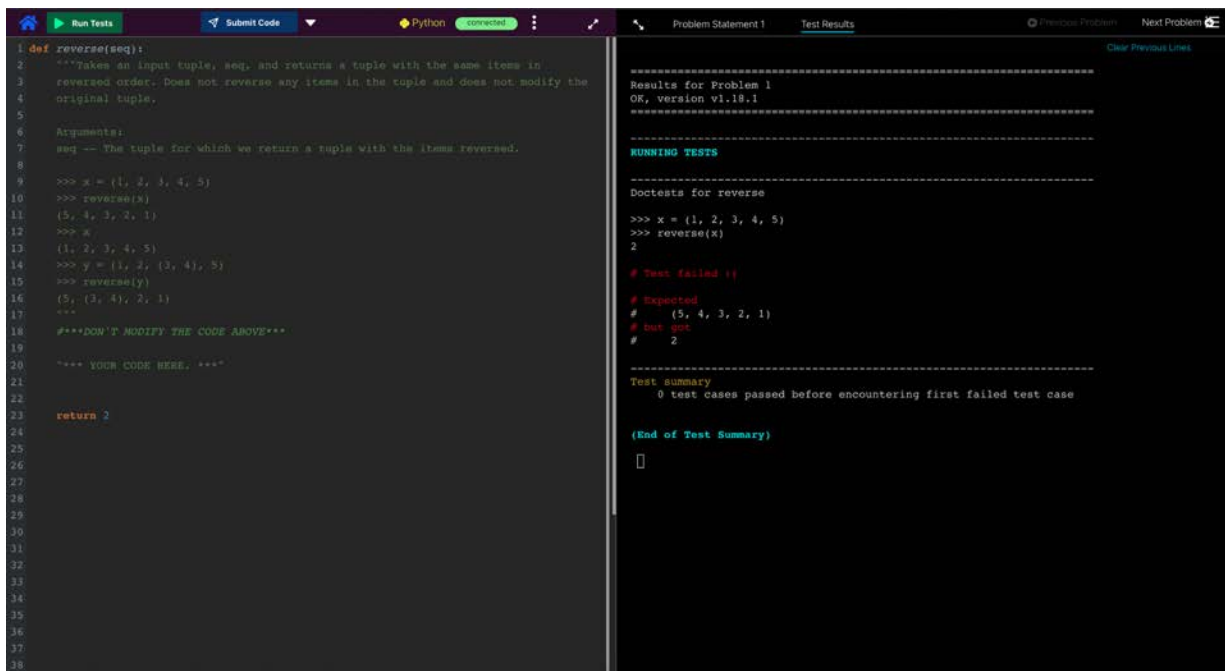


Figure 15: The tests output screen on the right side

When students click the “Run Tests” button (or whether they click “Test Results” tab explicitly), the right side of the screen changes from the Problem Statement to the “Test Results” display. If the code passes the tests, then the right side indicates so with the passed test cases in green; if

the tests fail as shown in the image above, then the right side shows the failed test cases in a red font. It is important to point out from a motivation standpoint how important it was to include the test results. Our initial idea was to hide the test results, and just to tell the student to code until they felt they had the right answer. However, we realized that students might stop early, thinking they were done -- but would have continued on to debug if they would have been given feedback that they hadn't crossed the finish line yet. .

The screenshot shows a web interface for a survey. At the top left, there is a logo for 'Berkeley EECS' and 'CODE PATTERNZ'. The user's email 'renaldo+33@berkeley.edu' and a 'Sign Out' link are visible at the top right. A sidebar on the left contains 'Introduction' and 'Question Bank'. The main content area is titled 'Survey' and features three text input fields with the following prompts: 'For the problems you were able to solve (i.e., all tests passed), what approach did you take to find the solution? Help us understand what was going through your mind while coding.', 'For the problems you were not able to solve (i.e., not all the tests passed), what do you feel were barriers that prevented you from finding the solution? What approaches did you try? (Remember it's okay if you weren't able to solve the problems.)', and 'Did you have any issues while using this application? Is there anything we could do to make it easier to use?'. 'Submit' and 'Close' buttons are located at the top right of the survey area.

Figure 16: The survey screen

After students completed the questions, they had to complete a survey, which was a way for students to highlight some of the strategies and setbacks they had while solving the problems. This qualitative feedback was really important for our study. For example, if we saw that a student took a very long time to complete a problem, we would also read their survey to understand what may have been the barrier that was preventing the student from solving the problem correctly.

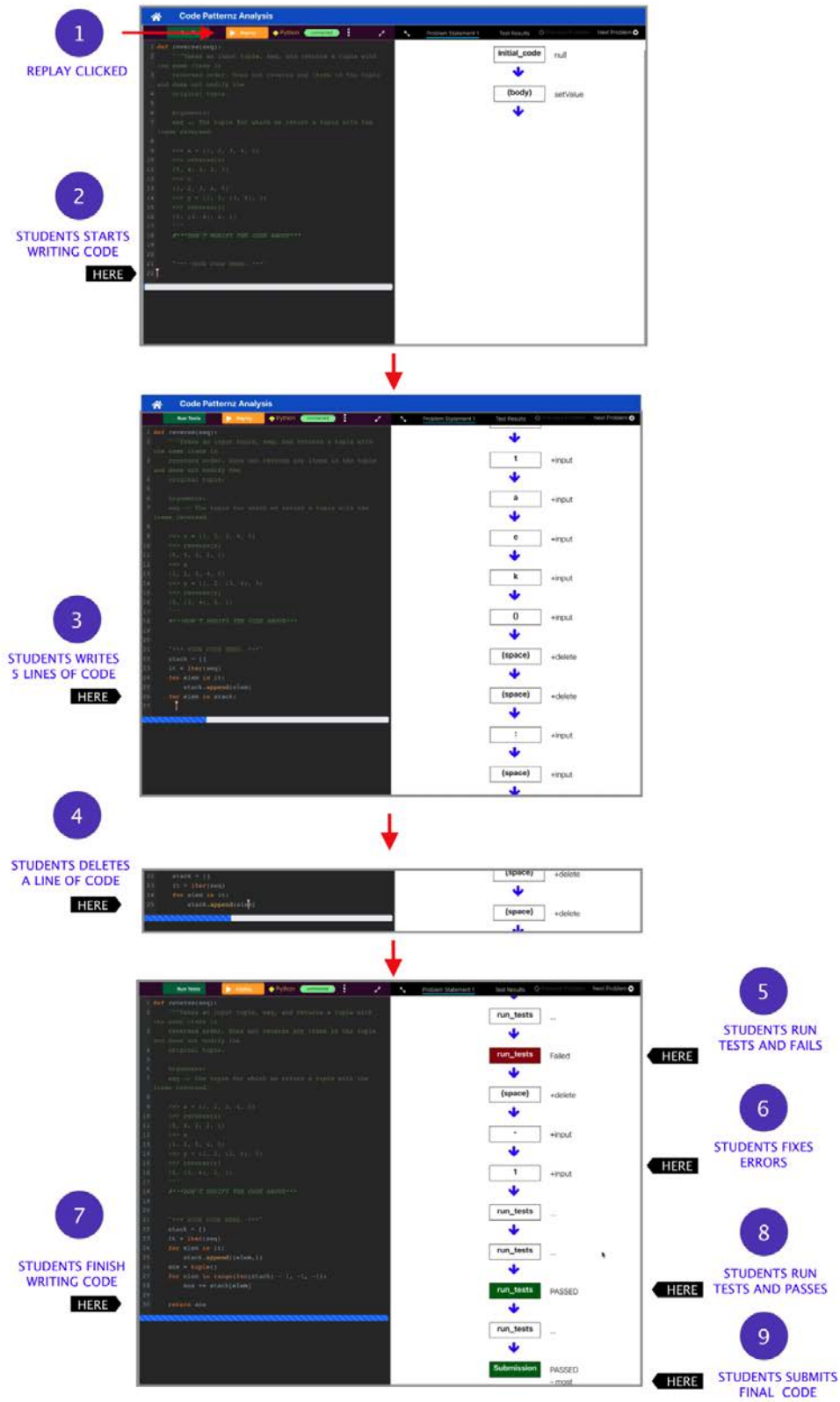


Figure 25: The replay sequence for a student

After the student has submitted their code, the researchers have the ability to replay it in order to visually analyze how the student solved the programming problem. In Figure 25, the *Replay* button is clicked and the editor starts to replay the students code as if it were being written live. As the figure shows, first the student starts to write their code, then adds more lines to their code, then deletes a line of code, then runs the test. After the test fails, the student then modifies their code several times to fix the error. The student then runs the tests again and their code passes the tests. Finally, the student submits their final code.

The process described above delineates many of the types of coding patterns that the researchers saw while observing different students' replay of the same programming problem. Many students would start by writing out an entire solution, then test their solution, to realize that their solution failed the test. Then the student would modify their code and run additional tests; this was an iterative process. After trying several times, most students would be able to solve the problem, but some would eventually give up and submit the code they thought was correct. Being able to witness this process was an important milestone and crucial indicator to the researchers that observing the process of how students write code to solve programming problems was useful.

5.3. Architecture

In this section, we discuss the different parts of the front-end and back-end of the web application. Since the back-end is more complicated, we start with an overview of the Flask application and database, then discuss Docker Containers, the front-end web application, and some components used for data analysis. Each of these main sections is numbered in Figure 17, showing the high-level architecture of the Code Patternz system.

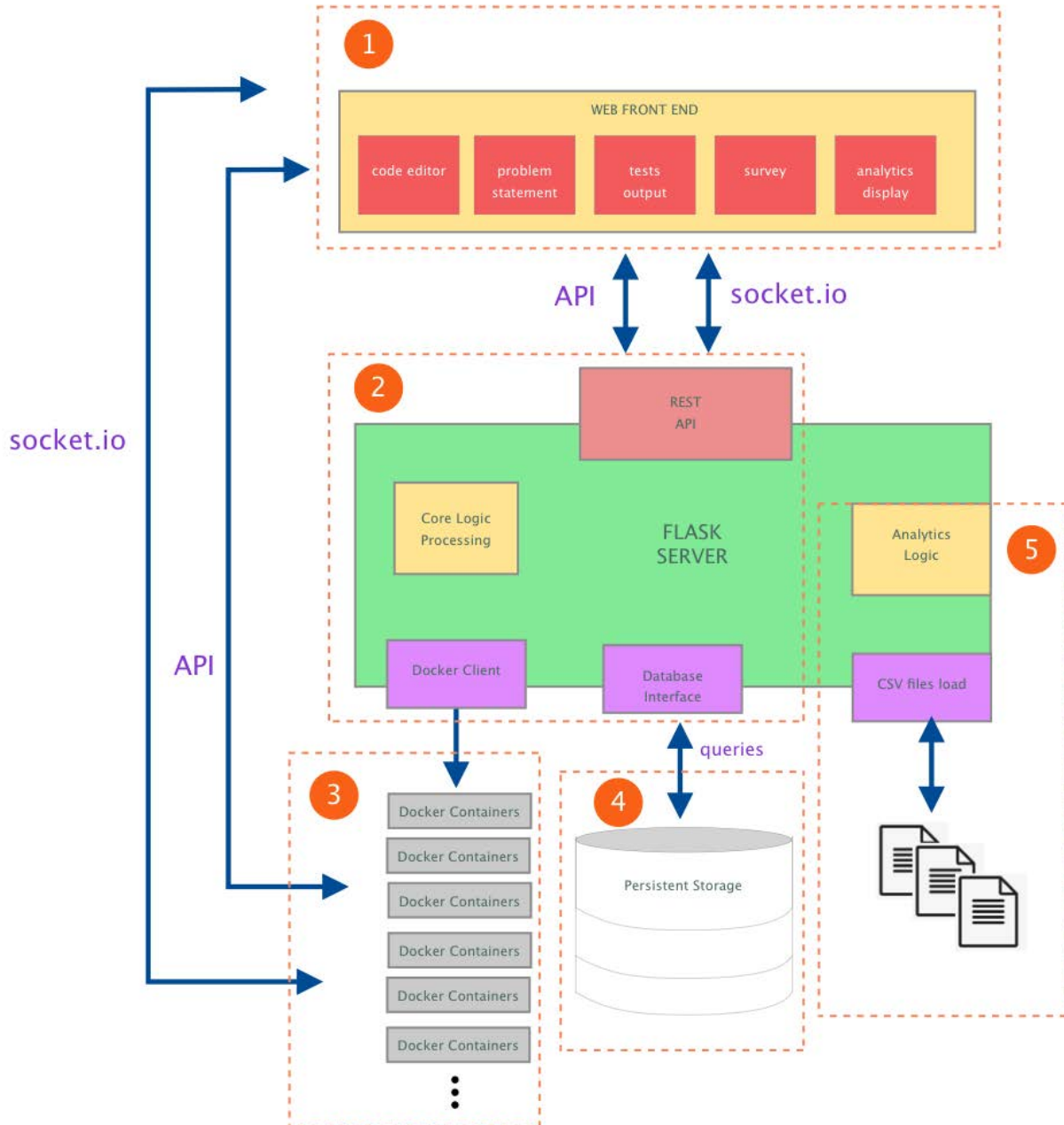


Figure 17: The Code Patternz high-level architecture.

5.3.1. Back-End Server

In order for the code editor to communicate with the back-end server, we use APIs that inform the back-end server of what functions to perform. The back-end code for Code Patternz was built using Flask. Flask is a framework that provides core basic services for coding the logic required to run the code editor. The reason Flask was chosen was because the framework is lightweight and helped us prototype various core features really quickly.

As can be seen in Figure 17, the Flask server is the central part of the entire system connecting the front-end, the Docker containers, the database, and the analytics engine. The front-end communicates with the back-end using a REST API service. The code editor (i.e., the *client*) makes *requests* to the Flask web *server*, which performs the appropriate action, then returns a response to the code editor indicating that the operation was carried out successfully.

For example, when users sign in (Figure 8), the front-end calls a specific Flask *route* named *signInCheck* (shown in Code Snippet 2), to check whether the students signed in using valid credentials. A *route* is the url the front-end uses to reach the back-end server.

```
@app.route('/api/signInCheck', methods=['GET', 'POST'])
def signInCheck():
    if request.method == 'POST':
        printer('Incoming..')
        theResponse = request.get_json()

        email = theResponse['email']
        passcode = theResponse['passcode']

        database = Database(p)
        users = database.getUserFromEmail(email)

        emailFound = False
        signInCodeCorrect = False

        user = []

        ...

        response.headers.add("Access-Control-Allow-Origin", "*")
        return response
```

Code Snippet 2: Sample route from Flask back-end

Most of the interactions with the front-end ends up calling a route on the Flask server. Each of the routes has their own specific function and is named in such a way that describes their function.

Another important task of the Flask server is to interface with the SQL database in order to store or retrieve information. This Flask server uses four basic operations: Create, Read, Update, and Delete [17]. After the Flask code initiates a database object, it can store or retrieve information. Below is an example of a database object, which gets the user's email from the database and checks whether the email is valid, and passes that decision to the front end so that the web application can show a message displaying if the email was found or not.

The interaction between the Flask server and the database is a very common transaction. It occurs when each key is pressed (or the cursor is moved), and the Flask server sends those characters (or cursor movements) to the database.

One of the main challenges with the Flask server was the heavy load generated from many students typing code at the same time, bombarding the server with many database store requests from all the code editors being used concurrently. Before running the full study on the entire CS61A class, we ran a pilot to make sure the application worked as intended. Though there were only 40 students in the pilot, many students experienced very long load times due to each student hitting the same Flask route at the same time. We realized that we had to put more resources in place to handle the large number of students who would be typing code at the same time. One of the important components was the addition of load balancers, discussed in the Deployment section.

5.3.2. Database

The Flask server interacts with a database that stores all the data for the Code Patternz web application. The benefit of using a database is that it stores all the data in an organized way.

As shown in Figure 17, circle 4, The Flask server issues queries to retrieve specific data from the database. Code Patternz relies on a SQL database, which is the most commonly used database for web applications. SQL databases were the most straightforward to use for this study because relational databases store data in tables, which nicely model the different parts of the web application. The number of columns in a table are fixed and the rows can grow to any size. The fact that the table rows can grow to any size was important for this study because storing data for each character pressed by each student would require hundreds of thousands, if not millions of rows being filled with data. For example, in order for us to capture user interactions to analyze patterns in student data later on, we had to create a database for storing the keys that were pressed, as well as the cursor movement on the code editor. This had to be tracked for every student that used the code editor.

Figure 18 shows a diagram of the relationship between the users table and the user_actions table in the database. The line that connects the two tables represents a relationship between the tables. The users table has a special column called the primary key, which holds a unique identifier for each row stored in the table. This is denoted by the **id** field in the users table (right side of Figure 18). In the table on the left, the user_id serves as the foreign keys, which reference the primary key of a row in the users table. These links between rows are called *relationships*, and are the foundation of the relational database model and how the Code Patternz web application organizes all its data. These tables allowed us to collect key click data for each student, and replay their code as if the student were typing it live. Note that there are many other tables within the database schema that follow the same *key-foreign key* model

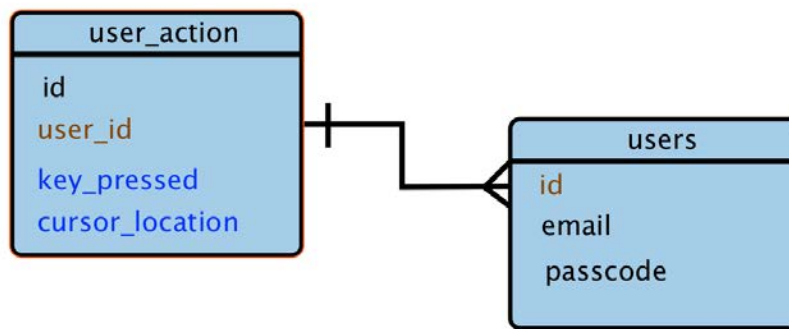


Figure 18: Example of the relationship between two tables in the Code Patternz database.

There were a number of factors that were important to us to consider when choosing a database framework. The first was ease of use. We wanted to make sure the database we chose had a good abstraction layer, that hid most of the complexity and exposed an easy-to-use API. The second was performance. Since we are storing many key presses per second, we wanted to make sure the database could handle the load. And finally, it needed to have Flask integration. Since Flask was the main framework we were using for our server, we wanted a database that was nicely integrated, so we wouldn't have to write the integration code ourselves. Our SQL database was able to meet all our needs for this study.

5.3.3. Docker

Another key component of the Flask server was its usage of Docker containers. Docker is an open source project for building, shipping, and running programs. It accomplishes this by using an operating system technology called *containers*.

For our study, the containers were used to run the Python code for each student inside an isolated Unix-like operating system environment. This was important so that their code could not break anything outside of the container instance it exists in. Another benefit of the docker container is that it contains all the dependencies necessary to run and test their Python code. This ensured that every student was running and testing their code in the same environment. As seen in Figure 17 (circle #3), the Flask server had to create many docker containers to handle running the code for each student. Each student *has their own unique container* created when they signed into the application because we wanted to isolate each student's code from other students.

In order to create the unique environment for each container that would run the students code, an *image* had to be created for the container. Container images are templates from which containers are created. The image is a stack of layers with each individual layer containing files and folders. In Figure 19 we show an example of the layers of the image that we had to create

for the Code Patternz code editor; these are combined to make up the environment that would run the code.



Figure 19: Sample image to run the Code Patternz container

For each container, we created an image called *Code Runner*. This was used to run the students' Python code and output the results of their tests, sent directly to the front-end, and shown in the code editor user interface.

Figure 20 below shows a flowchart describing how a container is created for each student who signs in. If a student signs in to a container that *was already created* for that student, the code editor would connect directly to that container instance. This prevents many containers from being created for the same student. If the student signs in and no container is found for the student, then a new container is created based on the Code Runner image. This ensured that each student is allocated a single, unique container to run their code.

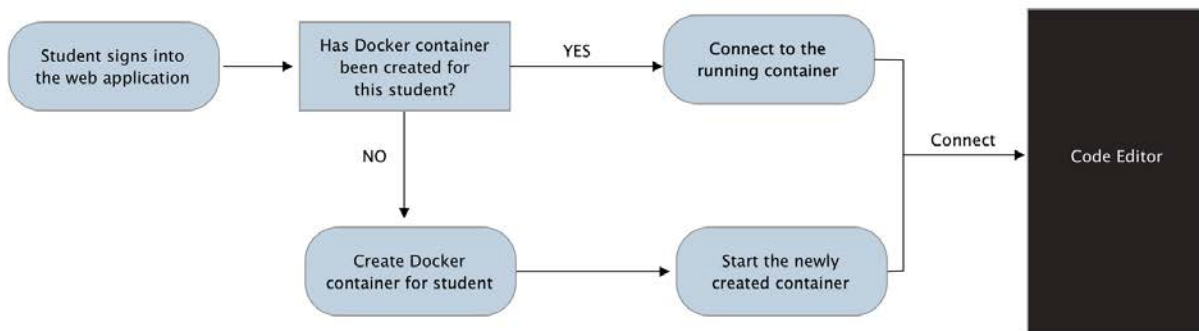


Figure 20: Diagram of the container creation process for a student.

As seen in Figure 17, the front end connects to the Flask instance running in the containers through a `socket.io` connection. `Socket.io` is a [TODO DESCRIBE IT], and it was needed as a bi-directional communication channel to allow messages from the Python interpreter to be sent directly to the front-end. The front end would then receive the message and display it on the user interface.

One of the challenges we experienced with docker containers was when errors occurred in the isolated environment. Since the container is a black box, the only way to monitor these errors was through log files. Some errors were very difficult to find because the log file description was very vague. In order to help debug these container errors, we added customized filtering and formatting of the log messages to help isolate the issue that may have occurred.

5.3.4. Front-End

One of the most important aspects of the front-end technology was that it used an architecture based on modern web technologies. Older web applications operate quite differently from modern web applications in that older web applications with multiple pages would refresh the entire page if any single item on the page needed to be updated. For example, Figure 14 shows the code editor on the left and the problem statement on the right. If the code editor was built using these older web page architectures, an update to *just one letter* in the problem statement would refresh both the code editor and the problem statement sections. This makes it feel like the web app navigates to a *whole different page*, but then shows the same page. We felt that this would be a less-than-stellar user experience to have the pages get torn down and redrawn, as shown in Figure 21.

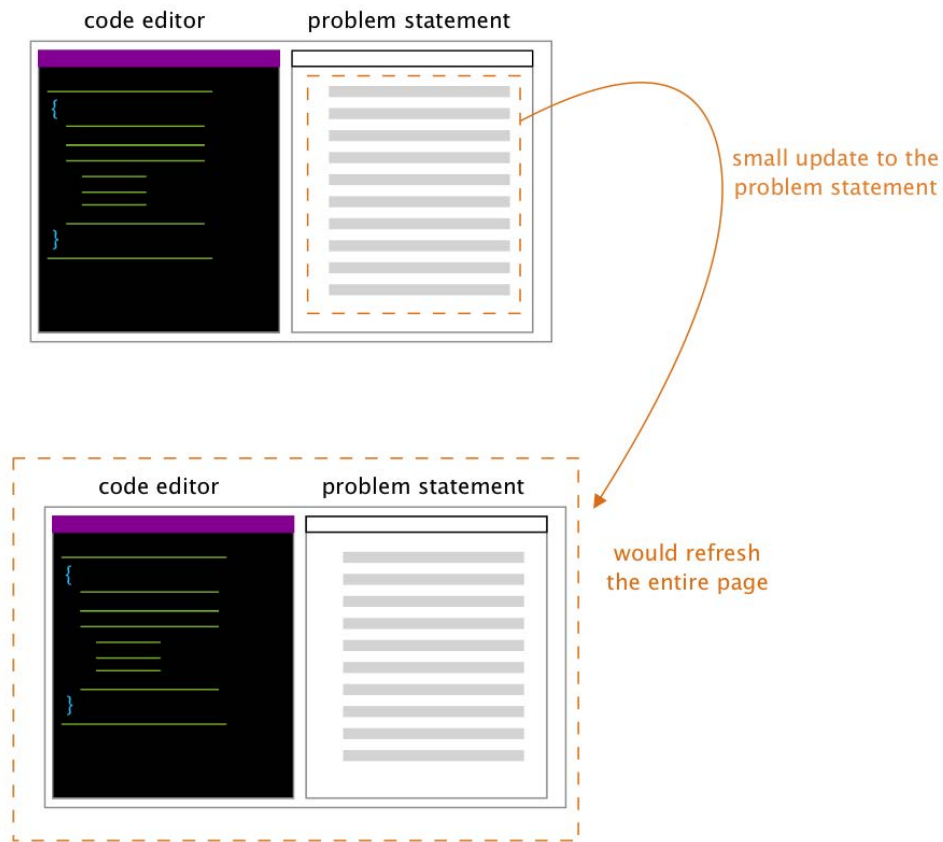


Figure 21: An illustration of a small update to the problem statement causing the entire page to be refreshed with older web technologies.

We wanted to adhere to a modern app paradigm known as a single-page app (SPA) model. In this model, the entire webpage does not reload for a single change to one area of the page. Instead, the application contains different sections that are loaded and unloaded into the same page itself. This was an important factor for us in choosing a front-end web technology because we wanted the code editor to feel as native as possible and less like a web application. As users interact with our code editor, we update the contents of the region with the data and HTML that matches what the student is trying to do. For example, if the student clicks the **Run Tests** button on the code editor while the problem statement is still visible, then only the region that shows the results of the tests should update. In this case, the problem statement would be hidden and the test results section would become visible.

This is shown in Figure 22 below, where the student clicks **Run Tests** and only the region that is surrounded by the dashed square on the right side of the page is updated to display the correct view. That view, the *Test Results* view, shows whether the student's code passed or failed the tests that we designate in the editor.

1) Run Tests

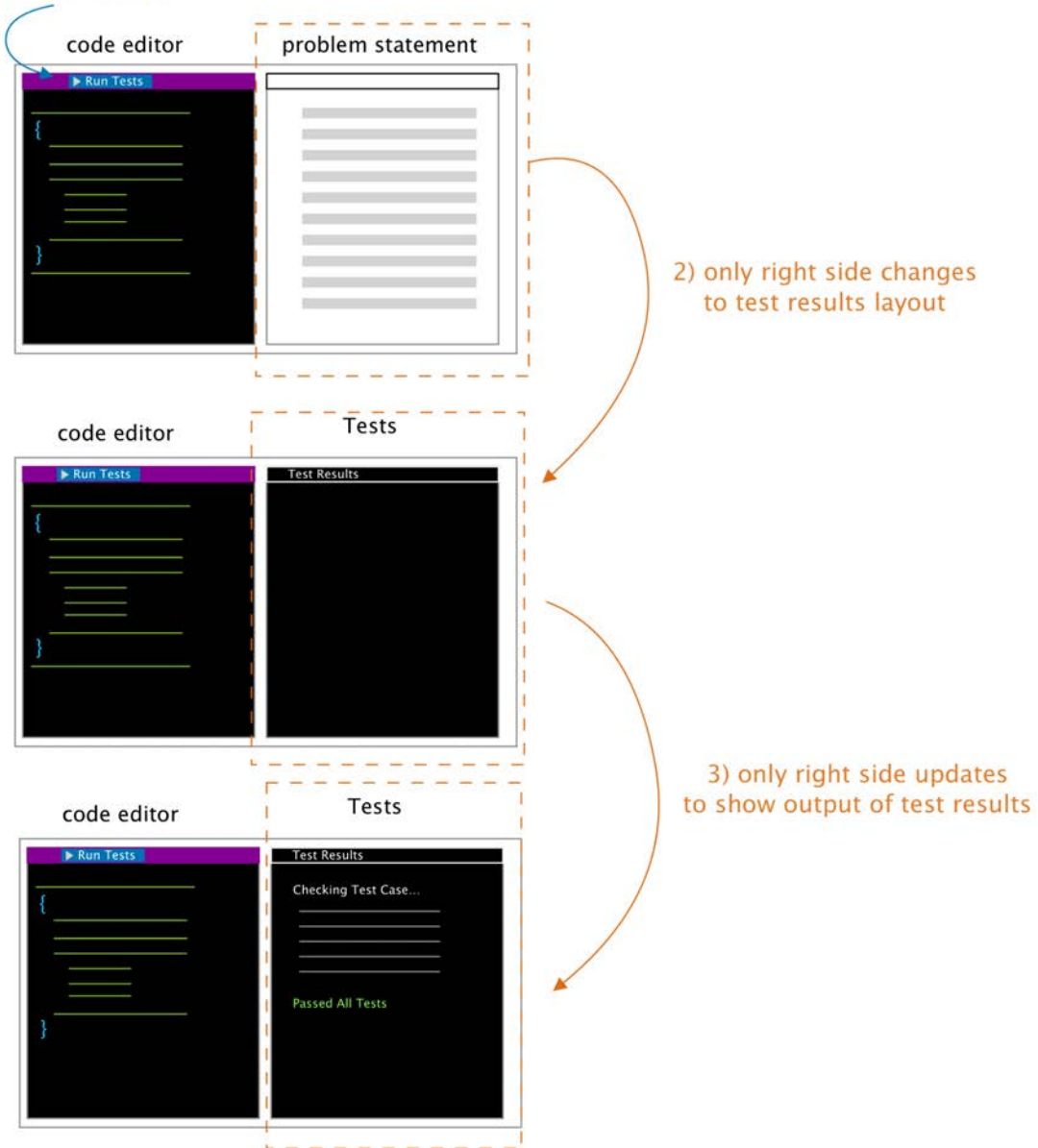


Figure 22: modern web-page architecture that updates a single region of the page instead of the entire page.

Overall, we felt that this resulted in a much more fluid experience for the students. In fact students would be used to this type of experience because popular web applications such as Gmail, Facebook, Instagram, or Twitter, all use the same single-page app. We decided to choose React as the front-end framework. React is created by Facebook and uses Automatic UI State Management [22]. It allows us to manage the different states of the Code Patterns web application and takes care of modifying only what needs to be changed during updates.

In Figure 23, we show an illustration of the Code Patternz React layout which was created to be modular and compact, like a tree. We break each of the main components into smaller components instead of treating the application as one monolithic application. Here the parent components represent an entire page, and the parents have children, which represent different regions on the page that are updated without affecting the other sections. For example, the Sessions page is shown in Figure 12. In Figure 23, the Sessions page is one the main pages where students can navigate to either the video instructions, the question bank, or to the survey. Within the diagram, the most important parent is the Code Page. This is the page where students will spend most of their time. The Code Page (parent) contains three main sections: the code editor, the problem statement, and the terminal sections. Updates to any of these regions will not affect other children regions on the page.

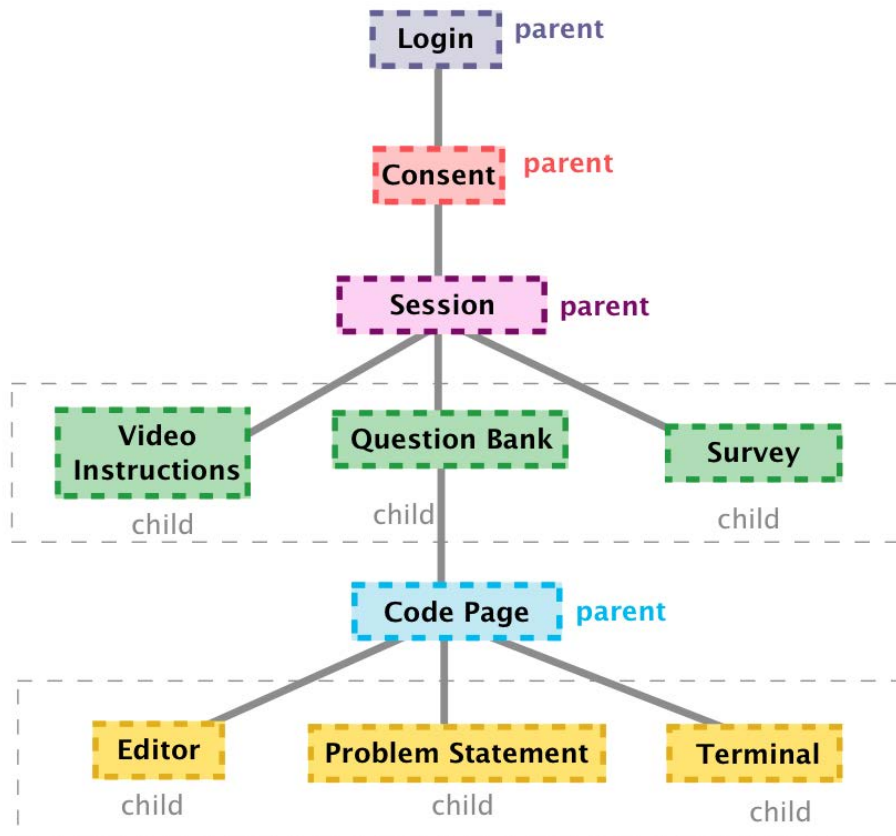


Figure 23: The layout of React components in the Code Patternz front-end web application

The *Editor* child is the region where students will type their code while trying to solve the programming problems. For our research, we track every character the students type, so any interaction with the editor triggers an event that sends the keyboard character to the Flask server, and eventually to the database for storage (shown in Figure 24). The Flask server also sends the student ID along with the character pressed, so that the database knows which student typed the character. There are additional columns in the table that are not shown, such as if the delete key were pressed, and the time that the key was pressed.

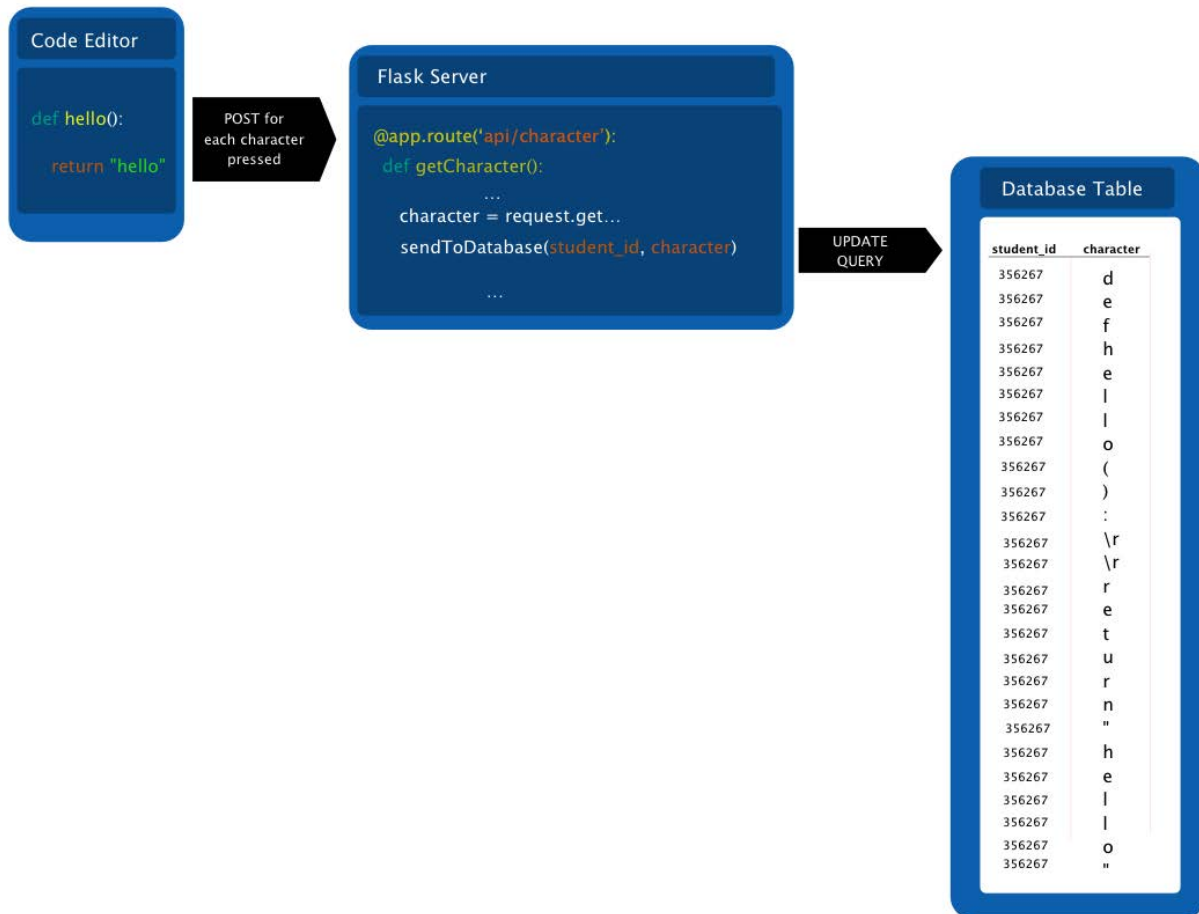


Figure 24: This is the sequence of events generated when characters are pressed in the code editor, sent to the server, and are stored in the database.

Figure 25 demonstrates how these stored characters are used to *replay* the students code on the user interface. Both the ability to replay, and to perform later data analysis, required us to store every character received.

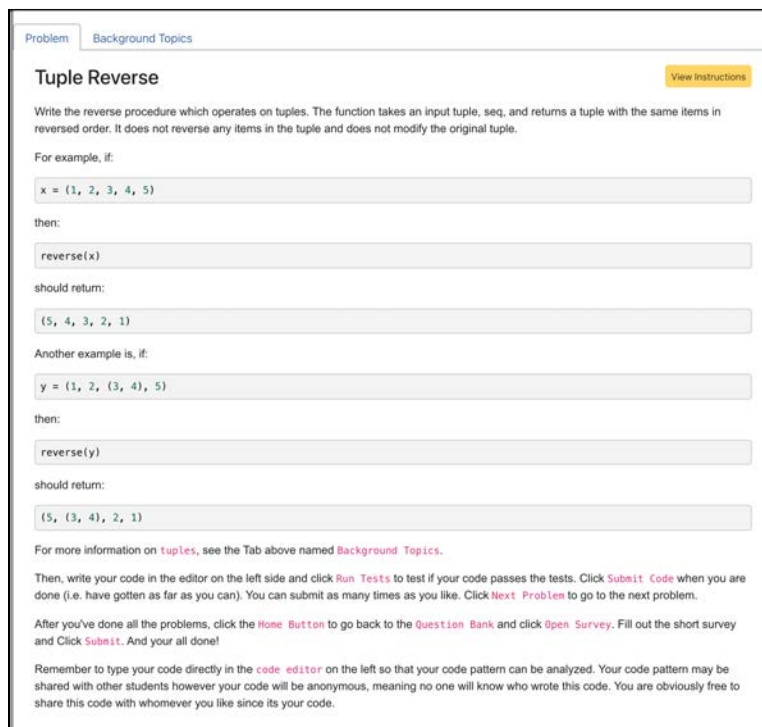
6. Data Analysis

Though the focus of this report is the Code Patternz web application, we felt it was important to show some basic analysis of the data that the tool was able to collect.

We begin with exploratory data analysis where we explain a histogram describing the number of characters typed overall for each problem. We then take a look at another more detailed histogram that describes the number of characters typed broken down by students who submitted a passed or failed submission for problem 1. Next we switch to analyzing a scatter plot, specifically looking at the number of characters typed by each student, broken down by the number of failed tests for problem 1. We then take a closer look at the actual code that students wrote from problem one and the patterns we observed.

6.1. Exploratory Data Analysis

One of the initial questions we had for problem one was: “how many students attempted that problem and what percentage of students got that problem correct?”. From our data, we saw that 269 students attempted problem one, and that 95.17% were able to solve the problem correctly. Figure 25 shows the written text for problem one.



The screenshot displays the 'Tuple Reverse' problem page. At the top, there are tabs for 'Problem' and 'Background Topics'. The title 'Tuple Reverse' is prominently displayed, with a 'View Instructions' button to its right. The instructions state: 'Write the reverse procedure which operates on tuples. The function takes an input tuple, seq, and returns a tuple with the same items in reversed order. It does not reverse any items in the tuple and does not modify the original tuple.' Two examples are provided. The first example shows an input `x = (1, 2, 3, 4, 5)` and the expected output `(5, 4, 3, 2, 1)`. The second example shows an input `y = (1, 2, (3, 4), 5)` and the expected output `(5, (3, 4), 2, 1)`. Below the examples, there are instructions on how to run tests, submit code, and navigate between problems. A note at the bottom mentions that code is shared anonymously for analysis.

Figure 25: Written text for problem 1

Another question that we had was: “how many characters do students type for each problem?”. We used a histogram to show the number of characters typed, and differentiated the percentiles by color so that we can see how many students fell above and below the 50th percentile. Since such a high percentage of students solved the problem correctly, this served as a gauge to see how many students knew how to solve the problem using 1) the least amount of code, 2) an average amount of code and 3) too much code. It served as a differentiator for which students found the problem easy and which students found it difficult.

Figure 26 shows a histogram describing the number of characters typed. We see that the 50th percentile is somewhere between 150 - 200 characters typed. We also see that the number of characters typed for the 25th percentile is between 0-50 characters while the number of characters typed for the 100th percentile is between 300 - 350 characters. This data suggests that students within the lowest 25% most likely found the problem the easiest to solve or had more experience with the correct syntax to use for this problem. Students at the long tail top of the histogram were most likely the students that struggled the most, and did not know the correct syntax to use.

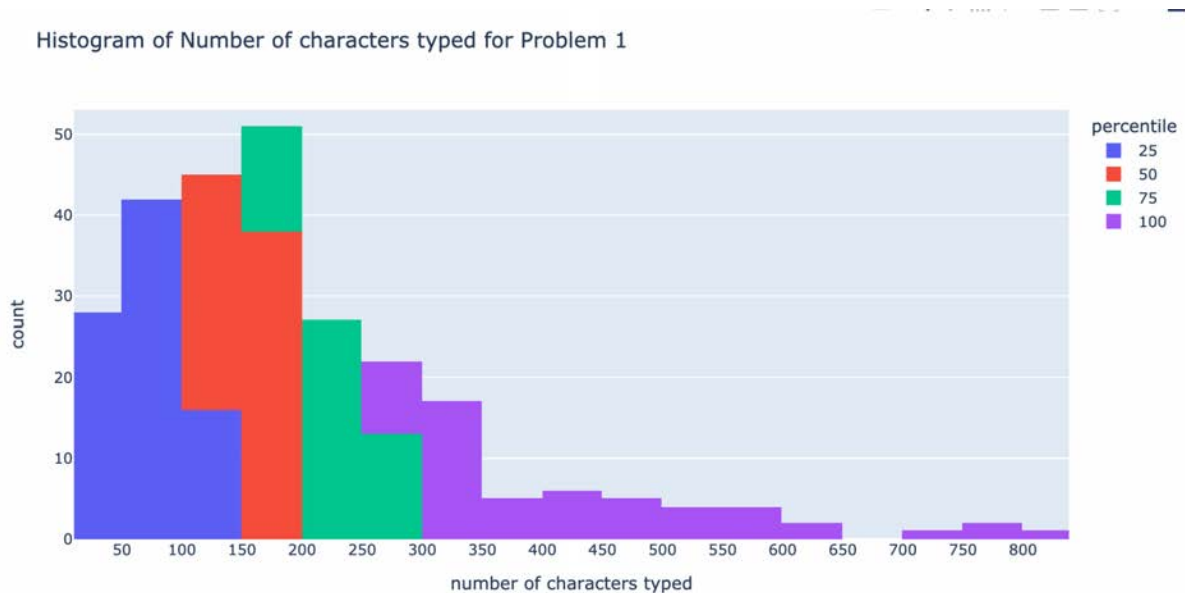
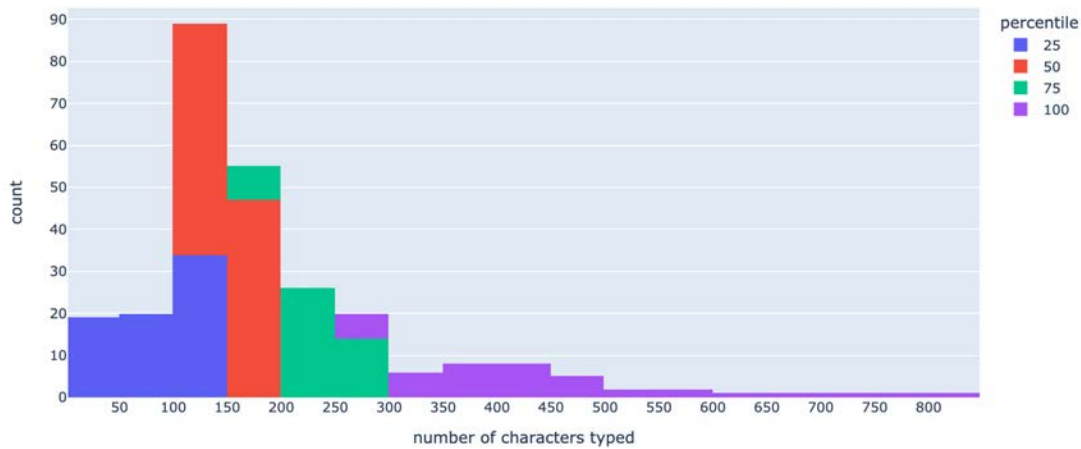


Figure 26: Histogram of the number of characters typed colored by percentile

For the sake of completeness, we show the histograms for problem 2 and 3.

Histogram of Number of characters typed for Problem 2



Histogram of Number of characters typed for Problem 3

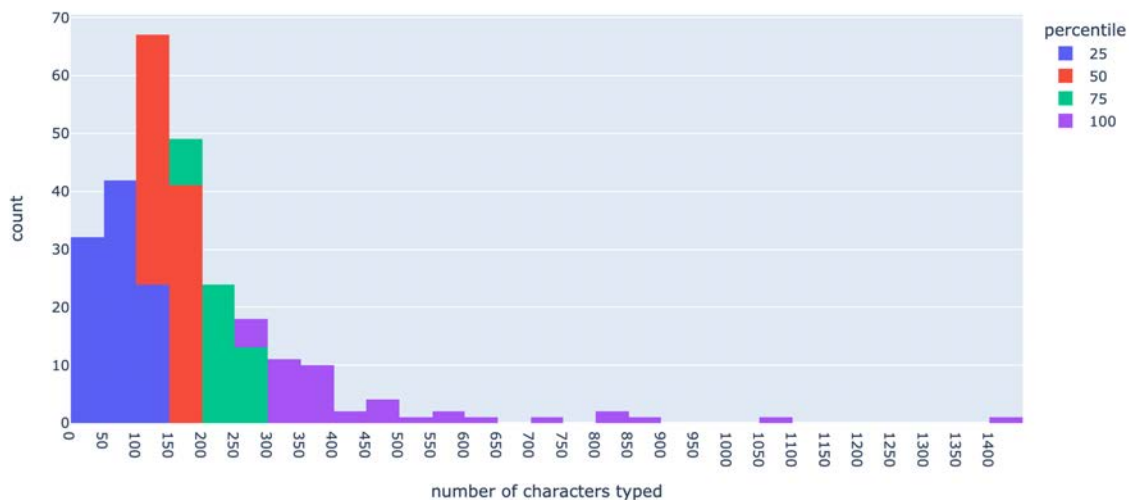


Figure 28: The top shows the histogram of the number of characters typed for problem 2 and the bottom shows the histogram for problem three.

The next query we had was based on the graphs above. We wanted to know, how does the data look if we separated the histogram by students whose final code submission passed all the tests and students whose final submission did not? In Figure 29, we show the histogram where the data is grouped into buckets of 50 characters. The green bars represent the students who submitted final code that passed all the tests and the red indicates students whose submitted code did not. As shown in Figure 26, we see that the range where the most characters are typed is between 150 - 200 characters. Since so many students were able to successfully solve this problem, the red bars are quite low across the entire histogram. The red bar from 0-50 is

higher because most of those students typed a small number of characters probably to test out the code editor but never finished trying to solve the problem. Overall this histogram does not tell us much important information, and problem two and three would have to be analyzed to see if students have a high number of failed submissions. If so, that would help us understand the differences between the failed and passed submissions based on the number of characters typed.

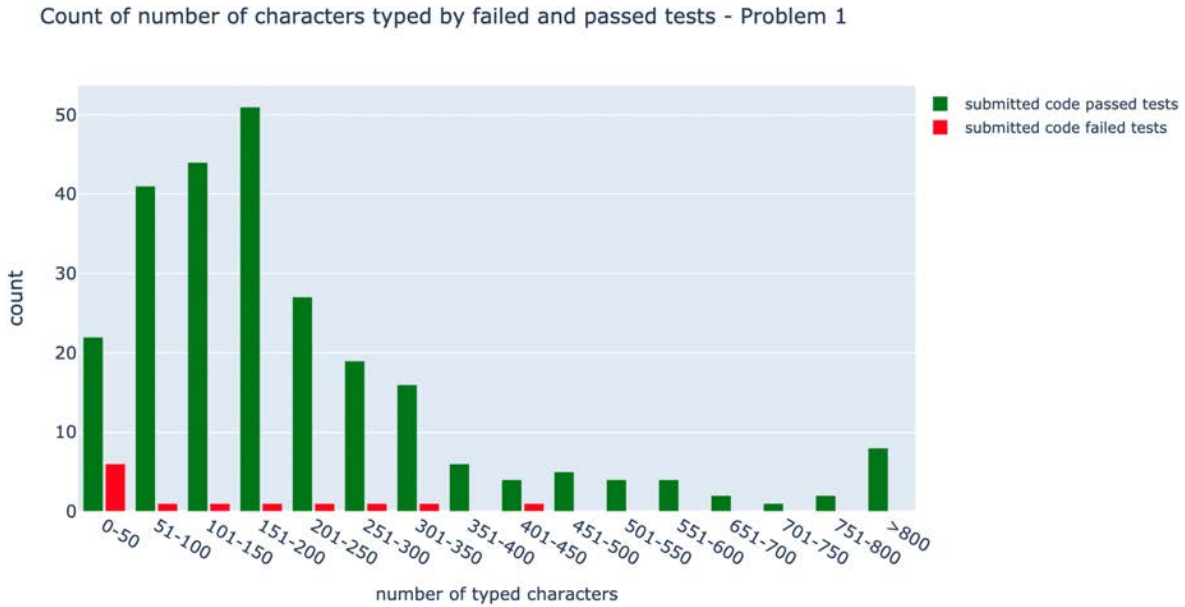


Figure 29: Graph showing the different buckets for the number of characters typed

We also wanted to know how many characters were typed compared to the number of times the student tests their code and the test fails? Figure 30 contains a scatter plot that helped answer that question, graphing the number of characters typed vs. the number of times the student tested their code and had it fail the tests. The dots on the scatterplot above represent students, and the color of the dot shows if the student's final code submission passed the test or not. We observed from the trend lines that the more code that students typed, the more likely they are to have a higher number of failed tests.

Num. Characters typed vs Test Failed count for Problem 1

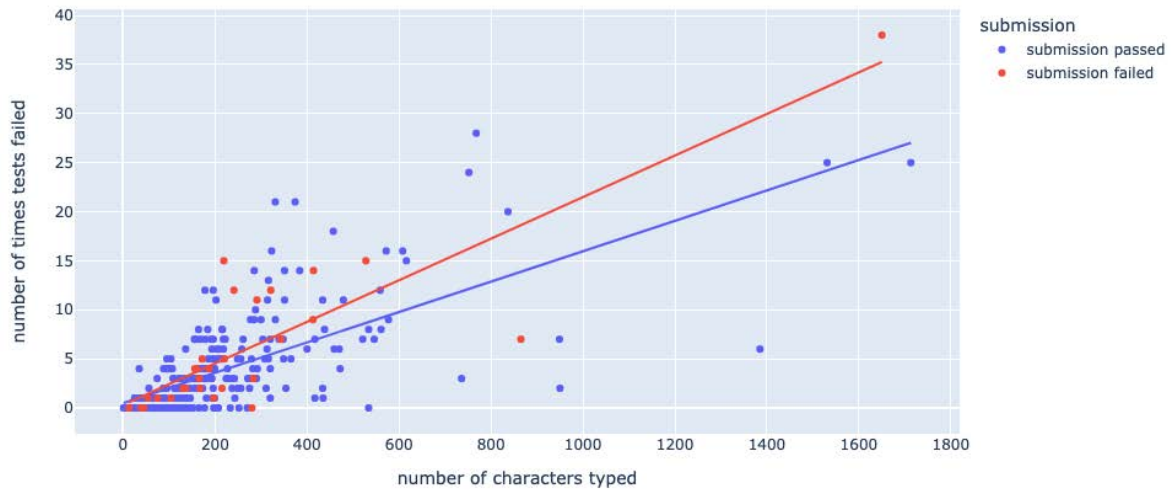


Figure 30: Shows a scatterplot of the number of characters typed compared with the number of times the students code failed the tests.

Usually, the process that students follow that lead to more characters and a higher number of failed test is as follows:

1. The student starts off the problem by writing some initial code,
2. The students test their code to see if the code passes all the tests.
3. The code fails to pass all the tests.
4. The student then then types more code to try another approach
5. The student run the tests again, and the tests fails
6. The student continues this process until they (hopefully) find a solution that passes all the tests

A logical conclusion from this pattern is that the more characters a student types, the more times they'll run tests, which unfortunately fail. However, there are only two students that never got it working, speaking to the resilience of these struggling students..



Figure 31: Figure showing two extreme points on the graph

An important pattern that we wanted to investigate more were the outliers. For example, in Figure 31, what type of code does the students in the furthest bottom left group write versus the student in the furthest top right group (see arrows). These are the students that wrote the fewest and most number of characters. Notice that, for the student in the top right corner, the final submission was not a correct solution. In the next section we take a closer look at some of the students' code to gain a better understanding of some of the data points in these graphs.

6.2. Student Code Patterns

Using the replay feature shown in Figure 25, we were able to observe the live replay of the students' code for problem one. This helped us capture how students took different approaches when trying to solve problem one. We were first interested in observing the typical solution for students within the group that wrote the fewest number of characters (25th percentile). Figure 32 shows the code for a student that fell within that group. We observed that the student was able to solve the problem with only one line of code using slicing syntax on the tuple.

The screenshot displays a coding environment with a Python code editor on the left and a test results panel on the right. The code defines a function `reverse(seq)` that returns a tuple with the items of `seq` in reversed order. The test results panel shows a sequence of steps: a series of 'run_tests' steps, followed by a 'Submission' step that is marked as 'PASSED'.

```

1 def reverse(seq):
2     """Takes an input tuple, seq, and returns a tuple with the same
3     items in
4     reversed order. Does not reverse any items in the tuple and does
5     not modify the
6     original tuple.
7     Arguments:
8     seq -- The tuple for which we return a tuple with the items
9     reversed.
10
11     >>> x = (1, 2, 3, 4, 5)
12     >>> reverse(x)
13     (5, 4, 3, 2, 1)
14     >>> x
15     (1, 2, 3, 4, 5)
16     >>> y = (1, 2, (3, 4), 5)
17     >>> reverse(y)
18     (5, (3, 4), 2, 1)
19     """
20     """DON'T MODIFY THE CODE ABOVE"""
21     """ YOUR CODE HERE. """
22     return seq[::-1]

```

The test results panel shows the following sequence of steps:

- run_tests ...
- run_tests ...
- run_tests ...
- run_tests PASSED
- run_tests ...
- Submission PASSED - most recent

```

22     return seq[::-1]

```

Figure 32: The top shows the entire user interface for a student who fell within the 25th percentile of characters typed. The bottom shows the students' solution enlarged.

From the survey from this student, the student

On the other hand, in Figure 33, where we observe a student within the 50th percentile, we see that more code was written, because the student instead uses a *for loop* and an additional list to store the items of the tuple in a reverse order. This solution passes the test (as can be seen on

the bottom left corner of Figure 27), however requires more code, and requires that an additional list be used.

The screenshot displays a coding environment with a Python code editor on the left and a test results panel on the right. The code defines a function `reverse(seq)` that takes a tuple and returns a reversed tuple. It includes docstrings, arguments, and test cases. The test results panel shows a sequence of inputs and actions: `(space)` +delete, `(space)` +delete, `s` +input, `e` +input, `q` +input, `(space)` +delete, `-` +input, `run_tests` ..., `run_tests` ..., `run_tests` PASSED, `run_tests` ..., and `Submission` PASSED - most recent.

```
1 def reverse(seq):
2     """Takes an input tuple, seq, and returns a tuple with the same
3     items in
4     reversed order. Does not reverse any items in the tuple and does
5     not modify the
6     original tuple.
7     Arguments:
8     seq -- The tuple for which we return a tuple with the items
9     reversed.
10
11     >>> x = (1, 2, 3, 4, 5)
12     >>> reverse(x)
13     (5, 4, 3, 2, 1)
14     >>> x
15     (1, 2, 3, 4, 5)
16     >>> y = (1, 2, (3, 4), 5)
17     >>> reverse(y)
18     (5, (3, 4), 2, 1)
19     """
20     #***DON'T MODIFY THE CODE ABOVE***
21     lst = list(seq)
22     result = []
23     for _ in seq:
24         result.append(lst.pop(-1))
25     return tuple(result)
26
27     """*** YOUR CODE HERE. ***"""
```

```
19     lst = list(seq)
20     result = []
21     for _ in seq:
22         result.append(lst.pop(-1))
23     return tuple(result)
```

Figure 33: The top shows the entire user interface for a student who fell within the 50th percentile of characters typed. The bottom shows the students' solution enlarged.

Above in Figure 31 we showed a scatter plot that highlighted a student whose number of characters typed was an outlier (top right corner). This student's code fell within the 100th percentile, which is the group of students who typed the most characters when trying to solve problem one. As we pointed out, that student was not able to solve the problem successfully which we discuss later.

Using the replay code feature shown in Figure 25, we were able to watch the replay of the students' code-writing process, and highlight where the student went wrong while trying to solve the problem. That student wrote over 1000 characters in attempting to solve the problem when the student from Figure 32 only wrote around 20 characters.

In Figure 34, you see the students' full code submission. We won't dive into the details because there is a lot that could be analyzed but one of the main reasons that the student typed so many characters is because the student wrote out notes to help solve the problem.



```
20
21     """ YOUR CODE HERE. """
22
23     def mango ( reversedTuple , remainingTuple ) :
24
25         if len(remainingTuple) == 0:
26             return reversedTuple
27
28         else:
29             return mango( (remainingTuple[0] , reversedTuple) ,
30                            seq[1:] )
31
32     #keep putting 1st element of remainingtuple at the beginning of
33     reversedtuple
34     #reversedtuple = () remainingtuple = (1 2 3)
35
36     #reversedtuple = (1) remainingtuple = (2 , 3)
37
38     #reversedtuple = (2 , 1) remainingtuple = (3)
39
40     #reversedtuple = (3 , 2 , 1) remainingtuple = ()
41
42     #final tuple returned = (3 , 2 , 1)
43
```

notes →

Figure 34: The notes that the student who struggled the most (and never quite managed to finish) used while solving the problem.

We discovered that one of the main reasons this student could not solve the problem was possibly because of their misunderstanding of a single element tuple. For example, as shown in

Figure 35, there is a very subtle syntactic difference between a string and a tuple of one argument:

```
a = ("s") is a string
```

and

```
a = ("s",) is a tuple with one element.
```

trailing comma is a must

Figure 35: Correct syntax for a tuple

Figure 36 shows the student's code; in the notes we indicate that the student tries to return a single element tuple using the incorrect syntax.

```
22
23 def mango ( reversedTuple , remainingTuple ) :
24
25     if len(remainingTuple) == 0:
26         return reversedTuple
27
28     else:
29         return mango( (remainingTuple[0] , reversedTuple) ,
30 seq[1:] )
31
32     #keep putting 1st element of remainingtuple at the beginning of
33     reversedtuple
34     #reversedtuple = () remainingtuple = (1 2 3)
35
36     #reversedtuple = (1) remainingtuple = (2 , 3)
37     #reversedtuple = (2 , 1) remainingtuple = (3)
38
39     #reversedtuple = (3 , 2 , 1) remainingtuple = ( )
40
41     #final tuple returned = (3 , 2 , 1)
42
43
```

wrong syntax for single element tuple

Figure 36: Showing that the student may have a wrong understanding of a single element tuple.

The data analysis that we have performed in this section was a result of the Code Patterns tool's ability to capture the sequence of characters typed by students solving each of three programming problems. As we have shown in this section, useful information can be discovered by looking at histograms, scatter plots, and individual student code replays.

The goal of this section was to perform a basic overview of some of the analysis. In the future, we hope a more in-depth analysis of the student data collected for this study will be performed by the researchers of this study.

7. Future Work

The Code Pattenz tool was created to collect and analyze the programming process. Though the tool was able to perform its main functions for this study, there are many additional features that are needed to make the tool more complete and useful.

One of the most important improvements that is needed is enhancements to the code replay features. The code replay was one of the most complicated aspects of this tool, because it required every stored character to be replayed in a smooth sequence. We initially did not know if our algorithm to store the code sequences would work, however after learning how similar tools replay code and performing several proof of concept experiments, we were able to create a working version of the code replay. In order to make this feature more complete, the feature would need ways to control the code replay sequence. This includes, pausing, stopping, rewind, and scrubbing the replay sequence so that researchers can have more control over the code they are watching.

Another enhancement that is needed is better filtering capabilities when using the table that shows a list of student recordings in the replay user interface. Currently, the table only contains a list of students who submitted their code in a fixed order . Better filtering capabilities would allow more freedom in selecting the students of interest such as: a specific time range, or a specific test status (e.g., students that passed all tests).

In terms of data analysis, the data that was collected for this study can be analyzed using more advanced statistical methods such as machine learning and deep learning algorithms. One of the interesting aspects of this study is that the data that is collected and analyzed is text data. This opens the door for new research which uses Natural Language Processing algorithms and Abstract Syntax trees to analyze source code data.

Another one of the interesting ways that the data can be analyzed is by incorporating the usage of Abstract Syntax Trees to further break down the components of the students code. Abstract Syntax Trees paired with a Markov model can help researchers capture common states that the students code transitions from and to, and even help predict hidden states using Hidden Markov models.

And finally, seeing that the core of this tool is a web-based code editor, it would be ideal for online competitive programming challenges. Not only would that entice more people to use the tool, but it would also be very interesting to study how experts solve certain programming problems to learn best practices.

8. Conclusion

In this paper, we describe **Code Patternz**, a tool created to record the programming process, so that we can find patterns in the ways students write code. Capturing this granular data is important because typically, a student's final submission does not provide insight into this type of valuable information -- the *process* (e.g., top-down, bottom-up, tests-first, etc.) and intermediate attempts are not stored.

Before we could capture any data related to a student's programming process, we had to build a custom code editor to do so. The Code Patternz code editor was designed to have a modern web application experience and was architected using various different technologies. These technologies included the React front-end framework, a Flask Server, Socket.io, a SQL database, Docker containers, load balancers, Amazon Web services, and many other resources.

In order to design a modern user experience, we studied the interaction design of other online code editors to see what types of interactions resembled that of a native code editor. We then performed usability testing and learned various points of improvements that needed to be made to the front-end user interface.

After building a stable tool with a modern user experience, we conducted the study during the summer of 2021 with students from a UC Berkeley course, CS61A. Over 269 students participated in the study by solving 3 programming problems in the Code Patternz code editor, which logged the keys that the students typed while solving the programming problems.

After storing the sequence of characters that the students typed, we utilized the Code Patternz replay features to watch the replay of how students wrote their code. The replay revealed interesting patterns such as code sequences where students were able to solve the problems with very little code or sequences where students typed many characters en route to trying to solve the programming problem.

We also perform some basic data analysis using histograms and scatter plots to see if there were any trends in the data. We saw one trend that the more characters students type, the more failed attempts they would have until the correct solution was found.

Finally, we suggested future opportunities for the Code Patternz software to grow, such as improvements to the replay features, the addition of progressive filtering of student submissions, and the application of Machine Learning to the data.

From this research, we show that it is possible to build tools that collect granular data on the programming process, and most importantly, that such tools do in fact provide useful insights into the patterns that students follow when solving programming problems.

9. Follow your inner voice

For this final chapter, I would like to conclude by sharing some important thoughts. When I first started as a graduate student at UC Berkeley in 2018, I remember my first interaction with a 4th year PhD student at the Welcome Weekend. She mentioned that she was doing research on some topic, and I asked her how one goes about doing research, because the process was foreign to me. She began talking about several resources graduate school students use, such as LaTeX and reading papers, and I had no idea what she was talking about, because I was so unfamiliar with the whole graduate school research process. It made me wonder if I belonged at graduate school at UC Berkeley, especially as the only African American male accepted into the Master's Program that year.

Fast forward three years and many struggles later, I am much more familiar with the research process and able to build a complex technical project and write a 60+ page Master Project report on it. I want to make it clear that a lot of what I learned at UC Berkeley was through making mistakes. I took easy classes that I probably should not have taken, and I took hard classes that I probably should not have taken. As someone going back to graduate school at the age of 36, I always wondered if I was too old to be back in school, but my hunger to become more knowledgeable about topics such as Machine Learning and Computer Science education, help provide a reminder that I am in the right place, at the right time in my life.

To anyone reading this and feeling like they are trying to find their way...my best advice is that no matter where you start from, follow your inner voice, and **listen to what it is telling you**. Many people told me that going back to grad school at my older age was a bad idea, but it turned out to be one of the best decisions I have made (both educationally and financially). I want to be very clear that I would not have my current position were it not for returning to graduate school at UC Berkeley at my older age.

I am an absolute believer that you have to make the hard decisions, and stick to it. No one can make it for you, and no one will come to save you. You have to be the one to save yourself and make your own life better. If that means going back to school at an age where you will be the oldest student in the class, or even as old as the Professor, (as was often the case for me), then so be it.

Sincerely,

Renaldo Williams

10. Appendix

10.1. Consent Form Human Subjects

<p>UNIVERSITY OF CALIFORNIA AT BERKELEY BERKELEY • DAVIS • IRVINE • LOS ANGELES • MERCED • RIVERSIDE • SAN DIEGO SAN FRANCISCO • SANTA BARBARA • SANTA CRUZ</p> <p>CONSENT TO PARTICIPATE IN RESEARCH <i>Analysis of Code Patterns in CS61A</i></p> <p>*Key Information*</p> <ul style="list-style-type: none">You are being invited to participate in a research study. Participation in research is completely voluntary.The purpose of the study is to gain a better understanding of the different ways that students solve programming problems in 61A and to understand common patterns in student code.You will be asked to solve 3 programming questions and allow researchers to record your responses to these questions as well as the keys you type while you answer these questions, and the length of time it takes for you to answer each question. You will also be asked to complete a 5-minute follow-up survey. Additionally, you may be asked to participate in a 15-minute Zoom interview.The study can take a total of 1.5 hours or less but depends on your hard or easy you find the programming problems, and whether or not you are invited and agree to participate in a Zoom interview.Risks and/or discomforts may include the risk of breach of confidentiality, however we are taking precautions to minimize this risk by following UC Berkeley security protocols.There is no direct benefit to you. However, the results from the study may benefit future 61A students and help current researchers in the field of Computer Science Education at Berkeley. <p>Introduction My name is Renaldo Williams, a graduate student at the University of California, Berkeley, working with my faculty advisor Professor Dan Garcia and with Professor John DeNero in the Department of Electrical Engineering and Computer Science (EECS). We are planning to conduct a research study, which we invite you to participate in. You are being invited to participate in this research study because you are enrolled in summer course 61A at UC Berkeley.</p> <p>Purpose The purpose of the study is to gain a better understanding of the different ways that students solve programming problems in 61A and to understand common issues that students run into.</p> <p>CPHS #2021-05-14344 Page 1 of 4</p>	<p>Procedures If you agree to be in this study, you will be asked to allow researchers to access your data as you solve 3 programming questions. You will be given extra credit for completing all three questions. PLEASE NOTE: You may complete all three questions for extra credit in this course regardless of whether or not you consent to participate in this research study. In other words, you may choose NOT to participate in this research and still receive extra credit for completing all 3 programming questions.</p> <p>The research procedures will consist of you using a web application to solve the questions, and while you type your code, the web application will log the keys you type so that the researchers can understand the path you take to solve the question. If you don't solve the question correctly, that is fine, it's more important for us to see the path you took while trying to solve the question.</p> <p>If you agree to participate in this research, the researchers will:</p> <ul style="list-style-type: none">Log your keys as you respond to the programming questions. In addition, the researchers will record your responses to the questions as well as the length of time it took for you to complete each question.Ask you to complete a short 5-minute online survey after answering the programming questions. <p>In addition, we may follow up with you to ask you to participate in a 15-minute follow-up Zoom interview. The interview will focus on your coding techniques or on a particular answer to the follow up survey. If you agree to the interview, we will also ask for your permission to video/audio record the interview.</p> <p>While working on the programming questions...</p> <ul style="list-style-type: none">Try to complete all 3 questions.Work on the questions as you normally would any programming question.Attempt the question by yourself without any help (human or through Googling the answers). If you get stuck, that is okay, you'll be able to skip that question after giving it a good try.Once you skip a question, you can go back if you like and continue working on the previous question.After completing the programming questions, you will be asked to complete a short 5-minute survey about how you approached the questions or where you got stuck. This survey is not part of your coursework for 61A; it is for research purposes only, and you will not get graded on it nor will you receive extra credit for it.Some students may have very interesting survey answers or programming techniques, and we may have follow-up questions. We will reach out to these students to see if they would be willing to have a 15-minute interview with us. This interview is not part of your coursework for 61A; it is for research purposes only, and you will not get graded on it nor will you receive extra credit for it. <p>Study time: Your study participation may take up to 1.5 hours. The time it will take for you to complete the 3 programming questions is approximately 1 hour, but will vary depending on how easy or difficult you find the questions to be. There is no additional time commitment for you to allow researchers to access your data from completing the programming questions. Again, please note that you may choose NOT to participate in this research and still complete these 3 programming questions for extra credit in course 61A. Participation in the survey will last approximately 5 minutes. You may also be asked to participate in a 15-minute Zoom interview with the researchers.</p> <p>CPHS #2021-05-14344 Page 2 of 4</p>
<p>Study location: All study procedures will take place at your computer. Researchers will ask you to use a web application while solving the 3 programming questions. The survey will take place online, using a survey hosted in the application. In addition, if you are asked to participate in an interview, the interview will take place on Zoom.</p> <p>Benefits</p> <ul style="list-style-type: none">There will be no direct benefit to you from participating in this study. However, we will analyze the different paths students take to the solutions, and share de-identified, aggregate data back with the class. We believe you may benefit through that study – seeing that there are many ways to approach these problems.The results from the study may benefit future 61A students as well! There may be pitfalls that we unearth, and we'll encourage future instructors to warn students of them, and share the best practices of the students who did complete the questions successfully. <p>Risks/Discomforts</p> <ul style="list-style-type: none">Some of you may feel nervous with your keys being logged while using the web application to solve the programming questions but we expect most students won't mind and mostly likely won't notice any difference.Breach of confidentiality: As with all research, there is a chance that confidentiality could be compromised, however, we are taking precautions to minimize this risk. <p>Confidentiality</p> <ul style="list-style-type: none">Your Berkeley email address will be recorded as part of the study so that we can contact you if necessary (for instance, if we would like to invite you to participate in a survey). By Spring 2022, your email address will be removed from the data so that the data set will be de-identified.If you are contacted about an interview and agree to participate, your interview data – including video/audio recordings – will be encrypted. The video/audio data will be destroyed by Spring 2022, after it is transcribed.Your study data will be handled as confidentially as possible. If results of this study are published or presented, individual names and other personally identifiable information will not be used. <p>To minimize the risks to confidentiality, we will encrypt private information, limit access to study records by only the researchers.</p> <p>If the data set is used for future research, all identifiers (e.g., email) will already have been removed from the data set. Identifiers might be removed from the identifiable private information. After such removal, the information could be used for future research studies or distributed to other investigators for future research studies without additional informed consent from the subject or the legally authorized representative.</p> <p>Your personal information may be released if required by law. Authorized representatives from the University of California may review your research data for purposes such as monitoring or managing the conduct of this study.</p> <p>Alternatives If you choose NOT to participate in the research but agree to solve the 3 programming questions for extra credit only, then your personal information, the path you take to solve the programming questions, your</p> <p>CPHS #2021-05-14344 Page 3 of 4</p>	<p>responses to the programming questions, and the length of time you took to complete them will not be recorded. In other words, your data will not be collected or used as part of this research.</p> <p>Rights Participation in research is completely voluntary. You have the right to decline to participate or to withdraw at any point in this study without penalty or loss of benefits to which you are otherwise entitled.</p> <p>Questions If you have any questions or concerns about this study, you may contact Renaldo Williams at renaldo@cs.berkeley.edu or Professor Dan Garcia at dggarcia@cs.berkeley.edu.</p> <p>If you have any questions or concerns about your rights and treatment as a research subject, you may contact the office of UC Berkeley's Committee for the Protection of Human Subjects, at 510-642-7561 or subjects@berkeley.edu.</p> <p>*****</p> <p>Thank you for your consideration, and we hope that you agree to participate in order to help us collect quality data for our research study.</p> <p>CONSENT Do you consent to participate in this research?</p> <p>O YES, I consent to participate in this research, which includes:</p> <ul style="list-style-type: none">The investigators logging my keys as I respond to the programming questionsThe investigators recording my responses to the programming questionsThe investigators recording the length of time it took for me to complete each programming questionCompleting a short online surveyCompleting a 15-minute Zoom interview (maybe) <p>If you agree to participate, please download a copy of this consent form for your records.</p> <p>O NO, I do not consent to participate in this research. I understand that I may still complete the programming questions and receive extra credit.</p> <p>CPHS #2021-05-14344 Page 4 of 4</p>

10.2. Programming Problems

Problem 1

```
def reverse(seq):
    """Takes an input tuple, seq, and returns a tuple with the same items in
    reversed order. Does not reverse any items in the tuple and does not
    modify the original tuple.

    Arguments:
    seq -- The tuple for which we return a tuple with the items reversed.

    >>> x = (1, 2, 3, 4, 5)
    >>> reverse(x)
    (5, 4, 3, 2, 1)
    >>> x
    (1, 2, 3, 4, 5)
    >>> y = (1, 2, (3, 4), 5)
    >>> reverse(y)
    (5, (3, 4), 2, 1)
    """
    ***DON'T MODIFY THE CODE ABOVE***

    """ YOUR CODE HERE. """
```

Code Snippet 1: The skeleton code for problem number one

Problem 2

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

def square_linked_list(lst):
    """ Squares each number in lst and returns the sum of the resulting values

    >>> lst1 = Link(1, Link(2, Link(3, Link(4))))
    >>> square_linked_list(lst1)
    30
    >>> lst2 = Link(3, Link(5, Link(4, Link(10))))
    >>> square_linked_list(lst2)
    150
    """
    # ***DON'T MODIFY THE CODE ABOVE***

    """ YOUR CODE HERE. """
```

Code Snippet 2: The skeleton code for problem number two

Problem 3

```
def is_palindrome(s):  
    """ Takes an input string, s, and returns True if s is a palindrome  
    and False if not using recursion.  
  
    >>> pd = 'racecar'  
    >>> is_palindrome(pd)  
    True  
    >>> pd = 'driver'  
    >>> is_palindrome(pd)  
    False  
    """  
    ****DON'T MODIFY THE CODE ABOVE****  
  
    """ YOUR CODE HERE """
```

Code Snippet 3: The skeleton code for problem number three

10.3. Sample of Survey Results

Anonymous Student 1:

1) For the problems you were able to solve (i.e., all tests passed), what approach did you take to find the solution? Help us understand what was going through your mind while coding.

For the first question, I thought of using a for loop to solve but I then remembered that tuples are just like a list but cannot be mutated. So, I used a = <list>[::-1] which reverses the tuple.\nFor the second question, I used normal recursion to find the output. If the link is empty return 0, else return the square of first and recursively call the rest of the link list.\nFor the third question, I first used the same approach as question 1 but then I saw that I have to use recursion so I had a base case where if len is 1 then it gives True, else if the first element is the same as the last element then recursively call back is_palindrome and mutate the list to have the new list from 2nd element to 2nd last element.

2) For the problems you were *not* able to solve (i.e., not all the tests passed), what do you feel were barriers that prevented you from finding the solution? What approaches did you try? (Remember it's okay if you aren't able to solve the problems.)

Solved All!

3) Did you have any issues while using this application? Is there anything we could do to make it easier to use?

I think it sometimes takes a little longer for it to connect to the server. The interface is good. Adding an Environment Diagram and Interactive Interface will be helpful if stuck on a problem.

Anonymous Student 2:

1) For the problems you were able to solve (i.e., all tests passed), what approach did you take to find the solution? Help us understand what was going through your mind while coding.

For the first two problems, I knew how to solve it in the most basic ways. But for finals practice, I tried to use recursion (especially the first problem). The first one was easy to visualize for me, as I can see in my mind that I would need a loop to do such and such, a recursion to go to "a deeper layer of the same thing" and do the same thing. For the `is_palindrome`, I was stuck for a long while because I was thinking too linearly in the sense that "I must return the true or false" This gets me narrow sighted into trying to see if there's any other way to manipulate slicing, especially slicing the end (but there's only one way, and it conflicts with my base elif statement). Then I realized, because recurse is basically going "deeper into another changed version of the string" then I it clicked for me, as the if-elif-else will always stay the same, i can just let the recursion do the work.

2) For the problems you were *not* able to solve (i.e., not all the tests passed), what do you feel were barriers that prevented you from finding the solution? What approaches did you try? (Remember it's okay if you aren't able to solve the problems.)

I was almost not able to do the `is_palindrome`. I think the other problem is that I'm being 'lazy' and not "thinking out loud" as the trains of logic for doing the problem is all in my head. This was apparent in the second problem which is easier than the first. But because the idea was in my head, I mistakenly used an inner function (which also would've worked if i could've implemented it correct, i.e. correctly see how the values will get passed to and from. To not "being lazy" I would have written the logic out on paper, and maybe that will help me get unstuck sooner. (but sometimes even that don't really work).

3) Did you have any issues while using this application? Is there anything we could do to make it easier to use?

I like this application, feels very much like a coding interview.

Anonymous Student 3:

1) For the problems you were able to solve (i.e., all tests passed), what approach did you take to find the solution? Help us understand what was going through your mind

while coding.

I think all of mine were recursive? I just realized that all of them could be solved by breaking the bigger problem down into a smaller version of itself - for example the palindrome you keep comparing the first and the last and move inwards, and if you find a pair that doesn't match its not a palindrome. I just knew how to solve it so I just had to implement it from there

2) For the problems you were *not* able to solve (i.e., not all the tests passed), what do you feel were barriers that prevented you from finding the solution? What approaches did you try? (Remember it's okay if you aren't able to solve the problems.)

I was able to solve all the problems

3) Did you have any issues while using this application? Is there anything we could do to make it easier to use?

none

11. References

[1] T. Jenkins, "On the Difficulty of Learning Programming," Loughborough University, United Kingdom, Aug. 2002.

[2] "Home : Occupational Outlook Handbook: : U.S. Bureau of Labor Statistics." <https://www.bls.gov/ooh/computer-and-information-technology/home> (accessed Aug. 19, 2021)

[3] A. Kavalchuk, A. Goldenberg, and I. Hussain, "An empirical study of teaching qualities of popular computer science and software engineering instructors using RateMyProfessor.com data," Jun. 2020, Accessed: Aug. 19, 2021. [Online]. Available: <http://dx.doi.org/10.1145/3377814.3381700>.

[4] N. Singer, "The Hard Part of Computer Science? Getting Into Class," *The New York Times*, Jan. 24, 2019.

[5] B. J. Keeton B. J. K. M. A. Brown, "7 Best Browser-based, Online Code Editors for Web Developers," *Elegant Themes*. <https://www.elegantthemes.com/blog/wordpress/7-best-browser-based-online-code-editors-for-web-developers> (accessed Aug. 19, 2021).

[6] themeselection, "Best Online Code Editors For Web Developers," *DEV Community*, Jun. 08, 2021.

[7] H. Meier, E. Tonisson, M. Lepp, and P. Luik, "Behaviour Patterns of Learners while Solving a Programming Task: an Analysis of Log Files," Apr. 2020, Accessed: Aug. 19, 2021. [Online]. Available: <http://dx.doi.org/10.1109/educon45650.2020.9125134>.

[8] P. E. Robinson and J. Carroll, "An online learning platform for teaching, learning, and assessment of programming," Apr. 2017, Accessed: Aug. 19, 2021. [Online]. Available: <http://dx.doi.org/10.1109/educon.2017.7942900>.

[9] T. Staubitz, H. Klement, J. Renz, R. Teusner, and C. Meinel, "Towards practical programming exercises and automated assessment in Massive Open Online Courses," Dec. 2015, Accessed:

Aug. 19, 2021. [Online]. Available: <http://dx.doi.org/10.1109/tale.2015.7386010>.

[10] J. L. Zachary and P. A. Jensen, "Exploiting value-added content in an online course," *ACM SIGCSE Bulletin*, vol. 35, no. 1, pp. 396–400, Jan. 2003, doi: 10.1145/792548.612016.

[11] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the effectiveness of error messages designed for novice programmers," 2011, Accessed: Aug. 19, 2021. [Online]. Available: <http://dx.doi.org/10.1145/1953163.1953308>.

[12] V. Mutiawani and Juwita, "Developing e-learning application specifically designed for learning introductory programming," Nov. 2014, Accessed: Aug. 19, 2021. [Online]. Available: <http://dx.doi.org/10.1109/icitsi.2014.7048250>.

[13] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein, "Modeling how students learn to program," 2012, Accessed: Aug. 19, 2021. [Online]. Available: <http://dx.doi.org/10.1145/2157136.2157182>.

[14] J. Noble, M. Homer, K. B. Bruce, and A. P. Black, "Designing Grace: Can an introductory programming language support the teaching of software engineering?," May 2013, Accessed: Aug. 19, 2021. [Online]. Available: <http://dx.doi.org/10.1109/cseet.2013.6595253>.

[15] M. Koorsse, C. Cilliers, and A. Calitz, "Programming assistance tools to support the learning of IT programming in South African secondary schools," *Computers & Education*, vol. 82, pp. 162–178, Mar. 2015, doi: 10.1016/j.compedu.2014.11.020.

[16] A. S. for P. Affairs, "Usability Testing," Nov. 13, 2013. <https://www.usability.gov/how-to-and-tools/methods/usability-testing.html> (accessed Aug. 19, 2021).

[17] J. Johnston, "What is a CRUD app and how to build one," *Budibase*, Jul. 06, 2021. <https://www.budibase.com/blog/crud-app/> (accessed Aug. 19, 2021).

- [18] S. Keidel, W. Pfeiffer, and S. Erdweg, "The IDE portability problem and its solution in Monto," Oct. 2016, Accessed: Aug. 19, 2021. [Online]. Available: <http://dx.doi.org/10.1145/2997364.2997368>.
- [19] D. Guzeev, "What is Emacs and Why You Should Use It? - Dmitry Guzeev," *Medium*, Jan. 18, 2018.
- [20] A. Luxton-Reilly, "Learning to Program is Easy," Jul. 2016, Accessed: Aug. 19, 2021. [Online]. Available: <http://dx.doi.org/10.1145/2899415.2899432>.
- [21] "10+ Best Online Code Editors to Use in 2021," *CatsWhoCode*, Feb. 16, 2020. <https://catswhocode.com/online-code-editor/> (accessed Aug. 20, 2021).
- [22] "Past, Present, and Future of React State Management – Lee Robinson," *Lee Robinson*, Feb. 02, 2021. <https://leerob.io/blog/react-state-management> (accessed Aug. 20, 2021).
- [23] Y. Kim and M. Lee, "Development of an unfolding model of procedures for programming learning of novice programmers," *Computer Applications in Engineering Education*, May 2021, doi: 10.1002/cae.22437.
- [24] "The Most Popular IDEs for Developers in 2021," *EPIC Software Development*, Apr. 14, 2021. <https://www.epicsoftwaredev.com/blog/the-most-popular-ides-for-developers-in-2021/> (accessed Aug. 23, 2021).
- [25] E. Lyulina, A. Birillo, V. Kovalenko, and T. Bryksin, "TaskTracker-tool," Mar. 2021, Accessed: Aug. 23, 2021. [Online]. Available: <http://dx.doi.org/10.1145/3408877.3432534>.