Live-coding vs Static Code Examples: Which is better with respect to Student Learning and Cognitive Load?

Adalbert Gerald Soosai Raj*

Department of Computer Science and Engineering
University of California, San Diego
gerald@eng.ucsd.edu

Arokia Xavier Annie R

Department of Computer Science and Engineering
College of Engineering Guindy, Anna University
annie@annauniv.edu

Pan Gu

Department of Math and Computer Sciences
University of Wisconsin-Madison
pgu7@wisc.edu

Jim Williams

Department of Computer Sciences
University of Wisconsin-Madison
jimw@cs.wisc.edu

Eda Zhang

Department of Curriculum and Instruction
University of Wisconsin-Madison
rzhang346@wisc.edu

Richard Halverson

Department of Educational Leadership and Policy Analysis
University of Wisconsin-Madison
rich.halverson@wisc.edu

Jignesh M. Patel

Department of Computer Sciences
University of Wisconsin-Madison
jignesh@cs.wisc.edu

ABSTRACT

Teaching programming using static code examples is the process of displaying pre-written code examples and explaining the purpose of the code. Live-coding is the process of writing code in-class in front of the students and thinking aloud while doing so. Static coding has more structure (similar to PowerPoint presentations) but lacks the authenticity and dynamic nature of writing code lively in front of an audience since the focus is more on the end product (i.e., program) rather than the process in which the program came to life. On the other hand, live-coding engages the students as the process is dynamic and makes the instructor's thought processes explicit to the students, but it lacks the structure and predictability of static code examples. We conducted an experiment in which we taught programming and data structures in C++ to two groups of undergraduate students. We used live-coding to teach one group (experimental) and static code examples to teach the other group (control). We conducted a pre-test and a post-test to measure students' understanding of programming before and after our intervention respectively. We collected a validated survey to measure the cognitive load experienced by the students in both the groups. Our experiments failed to show a difference between live-coding and static code examples with respect to student learning, but we

found that live-coding reduced the extraneous cognitive load on students when compared to static code examples.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**;

KEYWORDS

Live-coding, Static code examples, Programming pedagogy, Cognitive load theory, Cognitive apprenticeship

ACM Reference format:

Adalbert Gerald Soosai Raj, Pan Gu, Eda Zhang, Arokia Xavier Annie R, Jim Williams, Richard Halverson, and Jignesh M. Patel. 2020. Live-coding vs Static Code Examples: Which is better with respect to Student Learning and Cognitive Load?. In *Proceedings of Proceedings of the Twenty-Second Australasian Computing Education Conference, Melbourne, VIC, Australia, February 3–7, 2020 (ACE'20)*, 8 pages.
<https://doi.org/10.1145/3373165.3373182>

1 INTRODUCTION

Programming is usually taught using a variety of instructional techniques including but not limited to traditional approaches like writing code on a black/chalk board or document camera, displaying static code examples in a PowerPoint presentation, and active learning approaches like live-coding, peer instruction, think-pair-share, Process Oriented Guided Inquiry Learning (POGIL), etc.

Two commonly used techniques among them are static code examples and live-coding. Static code examples are programs (code snippets) that are written by an instructor before a class and are used to explain how the code snippet works. Live-coding is the process of writing code examples during the class in front of the

*Work done while the first author was a graduate student at UW-Madison

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACE'20, February 3–7, 2020, Melbourne, VIC, Australia

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7686-0/20/02...\$15.00

<https://doi.org/10.1145/3373165.3373182>

students. While writing code, the instructor usually thinks aloud to make his/her thought process visible to the students [13].

Some benefits of static code examples include: code is fixed and does not change on-the-fly [7], code is written and tested before the lecture, code snippets could be shared before the class to the students so that they can follow along during the lecture, and could save some lecture time since code is pre-written. Some drawbacks of static code examples include: entire code snippet is shown to the students at once and so students might feel overwhelmed, the process of writing the code snippet is hidden from the students and so they might develop a false impression that code is usually written in the way it is presented (i.e., starting from top to bottom, without any errors) while it is usually not the case [16].

Some benefits of live-coding include: making the process of programming visible to the students [3], making the instructor's step-by-step thought processes explicit [14], helps students understand implicit skills like incremental coding [26], helping learners understand how to detect and correct bugs in programs (i.e., debugging) [26], make instructors more relatable as students see that instructors too make mistakes while coding [2], and holding students' attention better since live-coding is a dynamic process [27]. Some drawbacks of live-coding include: slowing down the instructor as live-coding is a time-consuming process [26] (which might be better from the students' perspective); making it difficult for the students to take notes since the program is constantly changing (e.g., code refactoring); making it hard for the students to follow along since there could be multiple context switches between different applications like an Integrated Development Environment (IDE), terminal, and Jupyter notebooks [7].

Prior research on live-coding [25–27] report on the effectiveness of this approach for teaching programming to novice programmers. In these prior works, students' preferences on live-coding vs. traditional pre-coded examples were collected and they show that novice programmers prefer live-coding over static code examples for learning programming [25]. Even though the student preferences of live-coding for teaching introductory programming are well-known, there has been very limited work that has been done on measuring the actual learning benefits between a live-coding and static code examples session [27]. Also, to the best of our knowledge there is no prior work on measuring the cognitive load experienced by students in a live-coding versus static code examples session.

In this study, we focus on evaluating the approach of teaching programming using live-coding and static code examples with respect to student learning measured using pre- and post-tests. We also evaluate these two approaches with respect to the cognitive load perceived by the students in the two groups. Our research questions are the following:

- (1) *Is there any difference between teaching programming using live-coding versus static code examples with respect to student learning?*
- (2) *Is there any difference on the cognitive load imposed on the students due to these two teaching approaches?*

Cognitive load theory [33] explains differences in learning outcomes via differences in cognitive load. It generally predicts higher cognitive load will reduce learning [23]. We hypothesize that when static code examples are presented to students, this may increase

their extraneous cognitive load (see Section 3.1) and therefore decrease their learning since the entire code is presented at once. We believe that when pre-written code is presented to the students their cognitive load may be high because they might not be sure of where to focus in the code and so might feel overwhelmed. On the other hand, we believe that if code is written line-by-line as performed in a live-coding session, then the cognitive load on students might be less because they know exactly where to focus and hence this might increase their learning.

The contributions of our work are the following:

- (1) Provide evidence to show if one among the two approaches (i.e., live-coding vs static code examples) is better than the other with respect to student learning.
- (2) Provide evidence for the amount of cognitive load imposed on the students by both these teaching approaches and show which one might be better.

2 RELATED WORK

Soosai Raj et. al. conducted an experiment to study the effect of a students' native language for learning programming [29]. As a part of their study, they conducted a live-coding session to teach linked lists to both the experimental and the control groups. Surprisingly, they found that more than half of the students in both the groups have mentioned the usefulness of live-coding sessions in their open-ended feedback. They conducted a grounded theory analysis to understand the role of live-coding in learning introductory programming [26]. They found that live-coding helps students understand the process of programming, learn debugging skills, and acquire good programming practices like incremental coding, commenting, indentation, etc.

Rubin compared the effectiveness of live-coding against static code examples in an introductory C++ course [27]. In the static code examples (control) group, the instructors never typed code in-class but instead viewed, compiled, and executed code examples during the classroom lectures. In the live-coding (experimental) group, instructors started with a blank screen and taught programming by typing, compiling, testing, and executing code in-class. He administered surveys and analyzed student final grades to assess the effectiveness of live-coding. He found that students in the live-coding group performed significantly better than the students in the static code examples group on projects (large coding assignments) while the static code examples group performed slightly better than live-coding group on exams although the difference was not statistically significant. The survey results showed that students in the live-coding group preferred code examples over PowerPoint slides more than the control group. Based on these results he concludes that live-coding is as good as if not better than static code examples.

Paxton discusses the role of live programming as a lecture technique [25]. In a survey collected from students in a live programming class, he found that students preferred live programming when compared to a traditional approach (i.e., static code examples). Based on student feedback and his own experiences, he reports that live programming is not only fun but seems to be an effective technique for teaching programming.

Students in an introductory programming course were exposed to three active learning techniques for learning programming, namely mini-lectures, live-coding and in-class coding. Using a student survey at the end of the course, the authors found that students preferred guided instructional techniques like live-coding more when compared to minimally guided techniques like in-class coding [30]. A limitation with this study is that the students' preferences were not correlated with their performance in the course.

Live-streamed programming is similar to live-coding where a developer broadcasts his/her coding session on an open source software project with other developers using streaming websites like YouTube or Twitch. Alaboudi and LaToza conducted an exploratory study to better understand the characteristics, motivations, and challenges in live-streamed programming [1]. They analyzed 20 hours of live-streamed videos and surveyed 7 streamers about their experiences. They found that streamers are motivated by knowledge sharing, socializing, and building an online presence, but face challenges with tool limitations and maintaining engagement with their viewers.

Morrison investigated whether auditory explanations of code result in improved learning performance over written explanations in an introductory programming class [21]. They developed online instructional material in one of the three formats: 1) auditory only explanations, 2) text only explanations, or 3) both auditory and text explanations. They used the cognitive load questionnaire [22] to measure the cognitive load in different types of interventions. They found that altering the modality (text, oral, both) of code explanations did not improve student learning as measured by retention and transfer questions. They concluded that even though many instructors for introductory programming perform live-coding, assuming that presenting the code in an overhead display and explaining the code is an effective pedagogical technique, we still have no evidence that this dual modality teaching technique is effective for students learning introductory programming.

We note that in the above study, Morrison uses the term "live coding" for presenting code on an overhead display and providing oral explanations alongside. In this paper, we use the term static code examples for presenting and explaining code that is pre-written and we use the term live-coding for writing code in a step-by-step manner and explaining the thought processes behind coding.

Most of the previous works [6, 14, 15, 28], except the controlled experiments by Rubin [27], evaluate the usefulness of live-coding either using student surveys which asks about their preferences towards live-coding or using student responses to questions like 'What do you like/dislike about live-coding'. Even though these prior works report the value of live-coding as a pedagogical strategy for teaching introductory programming, we have very limited data (only one experiment [27]) that actually compares the effectiveness of live-coding with static code examples with respect to student learning. Also, to the best of our knowledge, none of the previous works have focused on measuring the cognitive load imposed on students during a live-coding versus static code examples lecture. We consider our study to be an extension of Rubin's work applied to a data structures course and the first of its kind to measure and compare the cognitive load imposed on students in a live-coding versus static code examples lecture.

3 THEORETICAL BACKGROUND

The two theories that are relevant to our work are cognitive load theory and cognitive apprenticeship.

3.1 Cognitive Load Theory

Cognitive load is the load imposed on an individual's working memory by a particular learning task (e.g., problem solving, thinking, reasoning) [36]. *Cognitive load theory* focuses on the cognitive resources used during learning and problem-solving. It is a set of evidence-based learning principles that are proven to result in efficient instructional strategies [9].

Instruction can impose three types of cognitive load on students' memory: intrinsic load, extraneous load and germane load. Intrinsic load (IL) is defined as a combination of the innate complexity of the material and the learner's prior knowledge. A subject (e.g., programming) is considered to have a high intrinsic (cognitive) load if it has a high element interactivity. That is, learning the subject requires remembering and processing many things simultaneously [35]. The prior knowledge and expertise also play a major role in intrinsic load since experts chunk knowledge in different ways than novices and so experts use much less working memory than novices for the same cognitive task [5]. Extraneous load (EL) refers to load imposed on the working memory that is not directly helpful for learning. For example, when materials used for learning (e.g., code, slides) are not easy to comprehend, then the amount of memory resources used to understand them do not directly help with learning and so they contribute to the extraneous load imposed on the learner. [19]. Germane load (GL), on the contrary, refers to instructional features that were beneficial to the learning. Germane load is a desirable form of cognitive load as it helps with schema acquisition and automation which enhances learning [24]. Prior works have suggested that to optimize the learning, instructional designers should design instructional materials such that they reduce the extraneous load, minimize the intrinsic load, and free the remaining working memory for learning such that the germane load is increased [34].

3.2 Cognitive Apprenticeship

Apprenticeship is a model of learning in which an apprentice learns the skills needed to perform a task from a skilled master. In traditional apprenticeship, the master shows the apprentice how to do the task, observes the apprentice perform certain aspects of the task, and gives more and more responsibility to the apprentice until the apprentice can do the task on their own [10].

Cognitive apprenticeship is a model of education that is based on traditional apprenticeship but incorporates the elements of modern-day schooling like conceptual understanding [10, 11]. The primary focus of cognitive apprenticeship is to make the thinking of an instructor to be visible to the students [10].

Cognitive apprenticeships differ from traditional apprenticeships in many ways. For instance, in traditional apprenticeship, the process of carrying out a task to be learned is usually easily observable (e.g., tailoring clothes). In cognitive apprenticeship, one needs to deliberately bring the thinking to the surface, to make it visible [10]. For example, an instructor needs to explicitly convey the different

algorithmic/design decisions that they make at every step while writing code.

Cognitive apprenticeship helps students learn the implicit processes involved in programming like incremental coding, debugging, and decomposing a large problem into multiple functions [26]. For example, it is a common practice to write a few lines of code that perform a certain task (e.g., read from a file), test if that particular task works, and then proceed to the next task. This practice is called incremental coding and is an important skill while writing code since at any point in time we can be confident that the previous few lines of code (within the function we are writing) works as expected. It is hard to teach such a practice using static code examples as the students see all the code at once and would have no idea how the code was written. But in a live-coding session, it is relatively easier to teach incremental coding since the process of programming is exposed to the learners.

4 METHODOLOGY

In this section, we explain the methodology that we used to conduct our experiment and collect our data.

4.1 Participants

We conducted our research at an Engineering college in Chennai, Tamil Nadu. Our study was conducted for six weeks (weeks 2 to 7 in a 15-week semester) during July and August of 2018. We recruited 180 students from a course on 'Advanced data structures using C++'. We randomly assigned our participants into one of the two groups (of 90 students each), namely live-coding (experimental) group and static code examples (control) group. Our study was advertised as an optional class where students can learn the basics of C++ Standard Library. As it was an optional class, many students dropped out of the study as the weeks progressed. Also, as the topics were building on top of previously taught topics, generally, students who missed a few classes stopped showing up in the later classes. We ended up with 81 students in total (40 students from control group and 41 students from experimental group) who attended all the lectures and tests. The students were in their second year of college and had previously taken an introductory programming and a basic data structures course in C [18]. The course alongside which we conducted our intervention was the next course that students take, in which they learn advanced data structures using the C++ programming language [31].

4.2 Experimental Design

The experiment was conducted using a *randomized control group pre-test post-test design* [12]. In this design, the participants are randomly assigned to two groups. This was possible since our intervention was conducted as an optional special topics course outside the students' regular courses. With this randomized design, all conditions are the same for both the experimental and control groups, with the exception that the experimental group is exposed to a treatment (live-coding), whereas the control group is not.

Random assignment to groups equalizes groups on existing characteristics (e.g., prior knowledge in programming) and, thereby, isolates the effects of the intervention [12]. The same instructor taught the two groups during our intervention. The instructor has

taught previously using both live-coding and static code examples and is equally comfortable teaching using both these techniques.

4.3 Experimental Procedure

We performed the following activities in both the control and the experimental group during our intervention: pre-test, classroom lectures, cognitive load survey, and post-test. Attendance was taken for all the classes during our intervention.

4.3.1 Pre-test. A pre-test was conducted to determine the students' understanding of the basic concepts in programming. There was a total of 5 questions on the pre-test. The following are the topics for the questions: (1) C basics, (2) functions and recursion, (3) pointers, (4) linked lists, and (5) memory regions (stack vs heap). The pre-test was conducted for a total of 50 points, 10 points for each question. The pre-test questions were created based on the previous topics that students learned in the basic programming and data structures courses, in consultation with the instructors who taught those courses. The above mentioned topics were tested in the pre-test as they would give us a baseline for understanding the students' prior knowledge about their programming skills before our intervention. The complete pre-test can be found at this link: http://bit.do/cpp_pretest.

A sample pre-test question is shown in Figure 1.

2. Functions and Recursion! [10 points]

What is the output of the following code snippet?

```
#include <stdio.h>
int func(int n)
{
    if (n == 0)
        return 0;
    return (n % 10 + func(n / 10));
}
int main()
{
    int num = 12345;
    int result = func(num);
    printf("result = %d\n", result);
    return 0;
}
```

OUTPUT:

Figure 1: A sample pre-test question.

4.3.2 Classroom Lectures. Twelve classroom-based lectures, each 50-minute duration, were taught for both groups. The C++ Standard Library was taught in those twelve lectures. The following topics were discussed during our intervention: console input/output, file input/output, strings, vectors, sets, maps, iterators, and algorithms. The same topics were taught to both the groups.

The main differences between the lectures for the two groups were the following: lectures were taught using *static code examples* for the students in the control group. That is, code examples were written prior to the lecture and they were displayed using CLion [4] IDE (Integrated Development Environment) during the lecture.

The instructor went through the pre-written code examples line-by-line and explained it to students. No new code was written during the lecture. Code was compiled and executed after explaining the code example to show the output of the code to the students. Code modification was done only when students asked questions of the form “What will happen if we change this line of code like this?”

On the other hand, the lectures were taught using *live-coding* in the experimental group. The same code that was pre-written and displayed to the students in the control group was written line by line in the experimental group. The instructor was sharing his thought processes as he was writing each line of code. No pre-written code snippets were used while teaching the experimental group. Code was compiled and tested after writing a few lines of code and before proceeding to the next few lines of code. This was one of the main differences when compared to the control group, where the code was executed only after the entire code snippet was explained.

The classes were scheduled such that each topic was taught first to one of the two groups in an alternating manner. For example, the topic on strings was taught first to the live-coding (experimental) group and then to the static code examples (control) group, while the next topic on vectors was taught first to the static code examples (control) group and then to the live-coding (experimental) group. The number of lines of code in the code snippets ranged between 12 and 60 with an average of 40. All code snippets were covered within 50 minutes in both the groups without the need for any extra time.

A sample code snippet used to introduce Strings and I/O streams is shown in Figure2.

```
1 #include <iostream>
2 #include <string>
3
4 int main() {
5     std::cout << "Hello, World!" << std::endl;
6     std::string name;
7     std::cout << "Enter your name: ";
8     // cin reads only until a space character.
9     // std::cin >> name;
10    std::getline(std::cin, name);
11    std::cout << "Welcome, " << name << std::endl;
12    return 0;
13 }
```

Run string

/Users/gerald/CLionProjects/C++StdLib/cmake-build-debug/string

Hello, World!

Enter your name: Pan Gu

Welcome, Pan Gu

Process finished with exit code 0

Figure 2: A sample code snippet that was used in both live-coding and static code examples lectures. This code snippet was shown as is in a static code examples lecture while it was written from scratch in a live-coding lecture.

4.3.3 Cognitive Load Survey. We used the validated cognitive load survey introduced by Leppink et al. [19] (and later adapted for computer science by Morrison et. al. [22]) to measure the three different types of cognitive load: intrinsic load (IL), extraneous load (EL), and germane load (GL) in learning. The cognitive load survey is a 10-question cognitive load component survey particularly designed for measuring cognitive load in computer science education [22]. As indicated by Leppink et al. [19], Questions 1 through 3

Instructions: All of the following questions refer to the lecture that just finished. Please respond to each of the questions on the following scale by circling the appropriate number (0 meaning not at all the case and 10 meaning completely the case):

1. The topics covered in the activity were very complex.
2. The activity covered program code that I perceived as very complex.
3. The activity covered concepts and definitions that I perceived as very complex.
4. The instructions and/or explanations during the activity were very unclear.
5. The instructions and/or explanations were, in terms of learning, very ineffective.
6. The instructions and/or explanations were full of unclear language.
7. The activity really enhanced my understanding of the topic(s) covered.
8. The activity really enhanced my knowledge and understanding of computing / programming.
9. The activity really enhanced my understanding of the program code covered.
10. The activity really enhanced my understanding of the concepts and definitions.

Figure 3: CS cognitive load component survey was originally developed by Leppink et al. [19] and later modified for CS by Morrison et. al. [22].

measure the intrinsic load (IL); 4, 5 and 6 measure the extraneous load (EL); and 7 through 10 measure the germane load (GL). Note that in both IL and EL (questions 1 through 6) higher values suggest larger detrimental effects to learning, while the value of GL (questions 7 through 10) indicate the opposite. That is, higher value of GL suggests larger positive effects for learning.

We distributed and collected this cognitive load survey to measure the three types of cognitive load at the end of one of the lectures in both the groups. The lecture in which we conducted this survey covered the topics of sets and maps. 79 students attended this lecture in the control group and 71 students attended this lecture in the experimental group.

The CS cognitive load component survey is a 11-point Likert scale survey with the following values (0 - not at all the case; 10 - completely the case). There are 10 questions in the survey. The CS cognitive load survey is shown in Figure 3.

4.3.4 Post-test. A post-test was conducted at the end of our intervention. It covered topics from strings, vectors, maps, iterators, etc. All the questions on the post-test were based on the material taught during the classroom-based lectures. The post-test contained 18 questions and was for a total of 58 points. The post-test was a paper-based test where the students had to write their answers in free form. The complete post-test can be found at this link: http://bit.do/cpp_posttest.

A sample post-test question is shown in Figure 4.

2. There are a few mistakes in the following code snippet that tries to print all the elements in a vector. Can you point out the mistake(s) and correct them?

```
void printVector(std::vector<int> v) {
    std::vector<int>::iterator it;
    for (it = v.begin(); it <= v.end(); ++it) {
        std::cout << it << ", ";
    }
    std::cout << std::endl;
}
```

ANSWER:

Figure 4: A sample post-test question.

5 RESULTS

As stated in Section 4.1, the data below only includes test data of students who attended all the classes and tests. A total of 81 students among 180 students from both the groups attended all the classes and tests. Among those 81 students, 40 students were from the control group and 41 students were from the experimental group.

The mean of the pre-test scores and the post-test scores for the two groups are shown in Table 1 and Table 2 respectively.

Table 1: Mean of pre-test scores for the two groups

Group	N	Mean	Std. Dev.	Std. Error of Mean
Control	40	57	22.4	3.5
Experimental	41	56.5	21.3	3.3

Table 2: Mean of post-test scores for the two groups

Group	N	Mean	Std. Dev.	Std. Error of Mean
Control	40	62.9	17.7	2.8
Experimental	41	56.6	15.9	2.4

5.1 Analysis of Pre-test Scores

The assumptions for two sample t-test (e.g., nearly equal variances) were satisfied by our pre-test data. We performed an independent samples t-test to compare the pre-test scores between the control group and the experimental group and found *no significant difference* in pre-test scores between the two groups ($t(81) = 0.08$, $df = 79$, $p = 0.93$). This means that there is no statistically significant difference between the two groups with respect to their prior programming knowledge (with an alpha value of 0.05 for statistical significance).

As both post-test and gain scores are good indicators of student learning [8, 20, 37], we used both the post-test and the gain scores in our analysis below to measure student learning.

5.2 Analysis of Post-test Scores

The below assumptions for independent samples t-test were satisfied by the post-test scores data of both the groups.

- (1) The Central Limit Theorem (CLT) applies to each sample individually. i.e., pre-test data is normally distributed in both the groups as confirmed by the Quantile-Quantile (Q-Q) plots.

- (2) The pre-test scores of both the groups are random samples that are independent of each other.
- (3) The variances of the pre-test scores are approximately equal. (i.e., Standard deviation of control group ($s_1 = 17.7$; Standard deviation of experimental group ($s_2 = 15.9$; $s_1/s_2 = 1.11 < 1.5$).

The mean post-test score for the control group is higher than that of the experimental group as shown in Table 2. We performed independent samples t-test to compare the post-test scores between the control group and the experimental group and found *no significant difference* in post-test scores ($t(81) = 1.67$, $df = 79$, $p = 0.09$).

5.3 Analysis of Gain Scores

The mean of the gain scores (gain = post-test - pre-test) for the two groups are shown in Table 3. The mean gain for the control group is higher than that of the experimental group. The assumptions for independent samples t-test (see Section 5.2) were satisfied by the gain scores of both the groups. We performed a t-test to compare the gain scores between the two groups and found *no significant difference* in gain scores ($t(81) = 1.24$, $df = 79$, $p = 0.21$) between the two groups.

Table 3: Mean of gain scores for the two groups

Group	N	Mean	Std. Dev.	Std. Error of Mean
Control	40	5.93	21.6	3.4
Experimental	41	0.05	20.9	3.2

5.4 Analysis of Cognitive Load

We conducted the following analyses using our cognitive load survey: average factor scores, t-test, and Mann-Whitney U test.

5.4.1 Average factor scores. First, we used means and standard deviations to describe each type of cognitive load, as measured by validated cognitive load instruments [19, 22]: intrinsic load (IL), extraneous load (EL), and germane load (GL). To describe a series of grouped Likert-type questions that indicates one trait, using means and standard deviations would be appropriate [32]. The mean and the standard deviations for the 10 questions and the factor average for the three types of cognitive load are shown in Table 4.

Table 4: Average factor scores by group

Live-Coding (N=71)				Static Coding (N=79)		
Q	Avg	SD	Factor Avg	Avg	SD	Factor Avg
1	3.47	2.26	IL=3.54 SD=2.33	3.61	2.3	IL=3.52 SD=2.32
2	3.68	2.41		3.46	2.35	
3	3.46	2.37		3.51	2.34	
4	0.76	1.11	EL=0.72 SD=1.33	1.35	2.00	EL=1.25 SD=1.91
5	1.1	1.86		1.48	1.95	
6	0.32	0.67		0.92	1.77	
7	7.61	2.13	GL=7.70 SD=2.03	7.91	1.98	GL=7.95 SD=1.86
8	7.69	1.94		7.72	1.85	
9	7.59	2.12		8.03	2.01	
10	7.91	1.96		8.16	1.59	

5.4.2 T-test. Next, we performed independent samples t-test on each type of cognitive load between the two groups to examine whether there was any statistically significant difference between the two groups with respect to the amount of cognitive load perceived by the students. We found a *statistically significant difference* on the extraneous load imposed on students between the two groups ($t(150) = -3.01$, $df = 148$, $p = 0.002$). There was no significant difference in the intrinsic and germane load imposed on students between the two groups.

5.4.3 Mann Whitney U test. To further investigate any specific distinction between the two groups, we conducted Mann Whitney rank test on each individual question in the survey. To describe ordinal data, non-parametric analysis of individual Likert-type questions fits better than calculating the average [32]. Particularly, we conducted two-sided Mann-Whitney U test to compare results of each question in the cognitive load survey between the live-coding group and the static code examples group. The assumptions of Mann Whitney test (e.g., independent populations) was satisfied with our data.

The results of our analysis in Table 5 show that the two groups differed significantly in question 6 (Q6) but did not differ significantly in other questions. Question 6 was part of the measure of extraneous load. This result aligns with the findings from our t-test of different types of cognitive load (see Section 5.4.2).

Table 5: Test statistic and p-values for Mann Whitney U test

Mann-Whitney U Test		
Q	Statistic Value	P-value
Q1	2768.5	0.78
Q2	3027.0	0.48
Q3	2777.5	0.81
Q4	2498.0	0.16
Q5	2576.5	0.29
Q6	2412.5	0.04
Q7	2603.0	0.36
Q8	2819.5	0.93
Q9	2369.5	0.07
Q10	2725.5	0.66

6 DISCUSSION

6.1 Interpretation of Results

Our study tried to compare the effectiveness of teaching programming and data structures using live-coding versus static code examples with respect to student learning and cognitive load. We measured student learning using post-test scores and gain scores. We measured cognitive load using a validated cognitive load survey [22].

We did not find any statistically significant difference between the post-test and gain scores between the two groups. Therefore, our experiment fails to show any differences between teaching programming using live-coding and static code examples with respect to student learning measured via paper-based tests.

We found a statistically significant difference in the extraneous cognitive load imposed on the students in the two groups. The

extraneous load imposed on the students in the live-coding (experimental) group was significantly lower than the extraneous load imposed on the students in the static code examples (control) group. This shows that live-coding decreases the extraneous load on students' working memory. One possible explanation could be that as code is developed in a step-by-step manner during live-coding, students know exactly which line of code to focus on, and therefore the extraneous cognitive load (i.e., load imposed on the working memory that is not directly helpful for learning) on the students is low and the students are able to use most of their working memory to focus on the learning task. On the other hand, in static code examples as the fully completed version of a code snippet is shown to students during a lecture, the students could feel overwhelmed seeing a full program (around 40 lines of code), and may be focusing on some unimportant elements in the program (e.g., include statements, function declarations) that might not directly help them with their conceptual understanding, which in turn increases their extraneous cognitive load.

Our findings match the findings of Rubin [27] that the static code examples group performed better than the live-coding group on paper-based exams, even though the difference in performance was not statistically significant. We consider our study to validate these results that live-coding might not directly help with increasing the performance of the students on paper-based exams when compared to static code examples.

We hypothesize that the reason for the poor exam performance of students in live-coding group may be because of the fact that the code on paper-based exams is *static*. The students in the static group are accustomed to seeing static code examples (as given on exams) more often than students in the live-coding group.

Although the results of our study matches those of Rubin's study [27], we note that Rubin's study examines student grades on assignments, exams, the final project, and the overall average to assess the effectiveness of live-coding, while our study primarily focuses on only pre- and post-tests. Therefore, we need more controlled experiments to better understand the usefulness of live-coding and static code examples on student learning.

6.2 Limitations and Future Work

One of the major limitations in our study is that we did not measure the code writing abilities of students using a computer-based test. If we had done so, we would have been able to collect more valuable data on how live-coding and static code examples impact the learning of students with respect to writing code by breaking down a problem into multiple smaller parts. Future work could better assess students by measuring their problem-solving abilities (e.g., decomposing a problem into multiple smaller parts) while writing code by video recording how students solve coding questions similar to those given in coding interviews [17].

Another limitation with our study is that the number of participants who attended all the lectures and tests were less than half of the total number of participants initially enrolled in our study. The reason for low retention is because our study was conducted as an optional special topics course (alongside a regular C++ course) and therefore students had no incentive to attend our classes other than the joy of learning. Therefore, our study suffers from self-selection

bias where only the students who were genuinely interested in learning the C++ standard library were retained till the end of our study. Future work could conduct such experiments as part of regular classes to increase the validity of the data with respect to different types of students.

Finally, the amount of code examples covered in a live-coding session could be fewer than that could be covered in a static code example lecture as live-coding might take more time since typing and frequent compilations are involved. In our controlled study, we mitigated this issue by selecting the code examples such that they could be completed within a 50-minute lecture. Also, as the instructor of our study was well versed with live-coding, there were no issues with time management. Time management could be an issue for instructors trying live-coding for the first time and hence computing education researchers trying to replicate this study should be aware of this issue as it is one of the threats to validity.

7 CONCLUSION

Our study compared two methods of teaching programming, namely live-coding and static code examples, and found that teaching programming using live-coding is no different than using static code examples with respect to student learning measured using paper-based programming tests. We also found that live-coding reduces the extraneous cognitive load imposed on students when compared to static code examples, which might help students to use more of their working memory to focus on the learning task. We believe that more work has to be done in this area to truly understand the benefits and drawbacks of different teaching methods in programming and how it affects students' performance, cognitive load, perceptions, sense of belonging, etc. We believe that our work is one of the initial few steps towards that direction.

ACKNOWLEDGMENTS

We thank Dr. Manjula for letting us to work with faculty, staff, and students in India. We are grateful to Dr. Renugadevi, Dr. Shanmuga Priya, and Prof. Karthika Devi for encouraging their students to attend our optional C++ classes, for helping with proctoring our tests, and for all their invaluable feedback throughout our fieldwork. We are indebted to Mr. Devabalan and Ms. Devika, for all their help with logistics and planning during our experiments. We also thank our anonymous reviewers and ACE program chairs, Andrew and Claudia for their invaluable feedback on our work.

REFERENCES

- [1] Abdulaziz Alaboudi and Thomas D LaToza. 2019. An Exploratory Study of Live-Streamed Programming. *arXiv preprint arXiv:1907.05931* (2019).
- [2] Lecia J Barker, Kathy Garvin-Doxas, and Eric Roberts. 2005. What can computer science learn from a fine arts approach to teaching? *ACM SIGCSE Bulletin* 37, 1 (2005), 421–425.
- [3] Jens Brednens and Michael E Caspersen. 2005. Revealing the programming process. In *ACM SIGCSE Bulletin*, Vol. 37. ACM, 186–190.
- [4] Jet Brains. 2019. Clion. <https://www.jetbrains.com/clion/>. (2019). [Online; accessed 05-April-2019].
- [5] John D Bransford, Ann L Brown, Rodney R Cocking, et al. 2000. *How people learn*. Vol. 11. Washington, DC: National academy press.
- [6] Russel E Bruhn and Philip J Burton. 2003. An approach to teaching Java using computers. *ACM SIGCSE Bulletin* 35, 4 (2003), 94–99.
- [7] Charles H Chen and Philip J Guo. 2019. Improv: Teaching Programming at Scale via Live Coding. (2019).
- [8] Edward Roland Cimler. 1973. *Gain vs. posttest scores as predictors of future performance*. Master's thesis. Citesee.
- [9] Ruth C Clark, Frank Nguyen, and John Sweller. 2011. *Efficiency in learning: Evidence-based guidelines to manage cognitive load*. John Wiley & Sons.
- [10] Allan Collins, John Seely Brown, and Ann Holum. 1991. Cognitive apprenticeship: Making thinking visible. *American educator* 15, 3 (1991), 6–11.
- [11] Allan Collins, John Seely Brown, and Susan E Newman. 1989. Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. *Knowing, learning, and instruction: Essays in honor of Robert Glaser* 18 (1989), 32–42.
- [12] Dimitar M Dimitrov and Phillip D Rumrill Jr. 2003. Pretest-posttest designs and measurement of change. *Work* (2003).
- [13] Rex E Gantenbein. 1989. Programming as process: a “novel” approach to teaching programming. In *ACM SIGCSE Bulletin*, Vol. 21. ACM, 22–26.
- [14] Alessio Gaspar and Sarah Langevin. 2007. Restoring coding with intention in introductory programming courses. In *Proceedings of the 8th ACM SIGITE conference on Information technology education*. ACM, 91–98.
- [15] Nasser Giacaman. 2012. Teaching by example: using analogies and live coding demonstrations to teach parallel computing concepts to undergraduate students. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 1295–1298.
- [16] Petri Ihanola and Ville Karavirta. 2011. Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education* 10 (2011), 119–132.
- [17] Sandeep Jain. 2017. GeeksforGeeks: A computer science portal for geeks. <http://www.geeksforgeeks.org/>. (2017). <http://www.geeksforgeeks.org/>
- [18] Brian W Kernighan and Dennis M Ritchie. 2006. *The C Programming Language*. (2006).
- [19] Jimmie Leppink, Fred Paas, Cees PM Van der Vleuten, Tamara Van Gog, and Jeroen JG Van Merriënboer. 2013. Development of an instrument for measuring different types of cognitive load. *Behavior research methods* 45, 4 (2013), 1058–1072.
- [20] Frederic M Lord. 1956. The measurement of growth. *ETS Research Report* (1956).
- [21] Briana B Morrison. 2017. Dual modality code explanations for novices: Unexpected results. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 226–235.
- [22] Briana B Morrison, Brian Dorn, and Mark Guzdial. 2014. Measuring cognitive load in introductory CS: adaptation of an instrument. In *Proceedings of the tenth annual conference on International computing education research*. ACM, 131–138.
- [23] Greg L Nelson and Amy Ko. 2018. On Use of Theory in Computing Education Research. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 31–39.
- [24] Fred Paas, Alexander Renkl, and John Sweller. 2003. Cognitive load theory and instructional design: Recent developments. *Educational psychologist* 38, 1 (2003), 1–4.
- [25] John Paxton. 2002. Live programming as a lecture technique. *Journal of Computing Sciences in Colleges* 18, 2 (2002), 51–56.
- [26] Adalbert Gerald Soosai Raj, Jignesh M Patel, Richard Halverson, and Erica Rosenfeld Halverson. 2018. Role of Live-coding in Learning Introductory Programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. ACM, 13.
- [27] Marc J Rubin. 2013. The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 651–656.
- [28] Amy Shannon and Valerie Summet. 2015. Live coding in introductory computer science courses. *Journal of Computing Sciences in Colleges* 31, 2 (2015), 158–164.
- [29] Adalbert Gerald Soosai Raj, Kasama Ketsuriyong, Jignesh M Patel, and Richard Halverson. 2018. Does Native Language Play a Role in Learning a Programming Language?. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, 417–422.
- [30] Adalbert Gerald Soosai Raj, Jignesh M Patel, and Richard Halverson. 2018. Is More Active Always Better for Teaching Introductory Programming?. In *Learning and Teaching in Computing and Engineering (LaTICE)*, 2018. IEEE.
- [31] Bjarne Stroustrup. 2000. *The C++ programming language*. Pearson Education India.
- [32] Gail M Sullivan and Anthony R Artino Jr. 2013. Analyzing and interpreting data from Likert-type scales. *Journal of graduate medical education* 5, 4 (2013), 541–542.
- [33] John Sweller. 1994. Cognitive load theory, learning difficulty, and instructional design. *Learning and instruction* (1994).
- [34] John Sweller. 2010. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational psychology review* 22, 2 (2010), 123–138.
- [35] John Sweller, Jeroen JG Van Merriënboer, and Fred GWC Paas. 1998. Cognitive architecture and instructional design. *Educational psychology review* 10, 3 (1998), 251–296.
- [36] Tamara van Gog and Fred Paas. 2012. *Cognitive Load Measurement*. Springer US, Boston, MA, 599–601. https://doi.org/10.1007/978-1-4419-1428-6_412
- [37] Donald W Zimmerman and Richard H Williams. 1982. Gain scores in research can be highly reliable. *Journal of Educational Measurement* (1982).