Learning to Code *with* Natural Language Programming *powered by LLMs*

Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming (*CHI'23*)
Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, Tovi Grossman

How Novices Use LLM-based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment (*Koli Calling'23*)
Majeed Kazemitabaar, Xinying Hou, Austin Z. Henley, Barbara J. Ericson, David Weintrop, Tovi Grossman

Majeed Kazemitabaar
PhD Candidate
University of Toronto

majeed.cc
@MajeedKazemi

Raspberry Pi Foundation

Research Seminars
February 2024

Computer Science
UNIVERSITY OF TORONTO

Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming

Majeed Kazemitabaar
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
majeed@dgp.toronto.edu

Justin Chow
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
justinchow@dgp.toronto.edu

Carl Ka To Ma
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
cma@dgp.toronto.edu

Barbara J. Ericson
School of Information, University of Michigan
Ann Arbor, Michigan, USA
barbarer@umich.edu

David Weintrop
College of Education, University of Maryland
College Park, Maryland, USA
weintrop@umd.edu

Tovi Grossman
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
tovi@dgp.toronto.edu

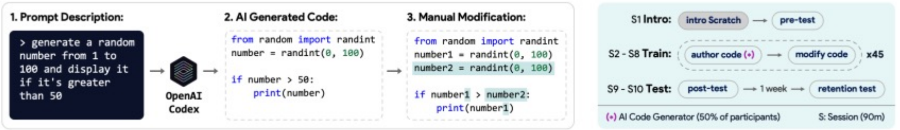


Figure 1: Left) Generate-modify usages with AI code generators. Right) Summary of our controlled study over 10 sessions.

ABSTRACT

AI code generators like OpenAI Codex have the potential to assist novice programmers by generating code from natural language descriptions, however, over-reliance might negatively impact learning and retention. To explore the implications that AI code generators have on introductory programming, we conducted a controlled experiment with 69 novices (ages 10-17). Learners worked on 45 Python code-authoring tasks, for which half of the learners had access to Codex, each followed by a code-modification task. Our results show that using Codex significantly increased code-authoring performance (1.15x increased completion rate and 1.8x higher scores) while not decreasing performance on manual code-modification tasks. Additionally, learners with access to Codex during the training phase performed slightly better on the evaluation post-tests conducted one week later, although this difference did not reach statistical significance. Of interest, learners with higher Scratch pre-test scores performed significantly better on retention post-tests, if they had prior access to Codex.

ACM ISBN 978-1-4503-9421-5/23/04...$15.00
https://doi.org/10.1145/3544548.3580919

CCS CONCEPTS

• **Human-centered computing → Interactive systems and tools**; • **Social and professional topics → Computing education.**

KEYWORDS

Large Language Models, AI Coding Assistants, AI-Assisted Pair-Programming, OpenAI Codex, GPT-3, ChatGPT, Copilot, Introductory Programming, K-12 Computer Science Education

ACM Reference Format:
Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23), April 23–28, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3544548.3580919

1 INTRODUCTION

Powered by the recent advancements in Deep Learning [88], Large Language Models that are trained on millions of lines of code, such as OpenAI Codex [14], can generate code from natural language descriptions (Figure 1, Left). In addition to enabling natural language programming, these AI coding assistants can perform numerous operations including code-to-code operations like code completion, translation, repair, and summarization, along with language-to-code operations such as code explanation and search [58, 79]. By generating code from simple sentences instead of formal and syntactically fixed specifications, these AI Coding Assistants may lower the barriers to entry into programming.

CHI Conference in Human Factors in Computing
Hamburg, Germany, May 2023

How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment

Majeed Kazemitabaar
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
majeed@dgp.toronto.edu

Xinying Hou
School of Information, University of Michigan
Ann Arbor, Michigan, USA
xyhou@umich.edu

Austin Henley
Microsoft Research
Redmond, Washington, USA
austinhenley@microsoft.com

Barbara J. Ericson
School of Information, University of Michigan
Ann Arbor, Michigan, USA
barbarer@umich.edu

David Weintrop
College of Education, University of Maryland
College Park, Maryland, USA
weintrop@umd.edu

Tovi Grossman
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
tovi@dgp.toronto.edu

ABSTRACT

As Large Language Models (LLMs) gain in popularity, it is important to understand how novice programmers use them and the effect they have on learning to code. We present the results of a thematic analysis on a data set from 33 learners, aged 10-17, as they independently learned Python by working on 45 code-authoring tasks with access to an AI Code Generator based on OpenAI Codex. We explore several important questions related to how learners used LLM-based AI code generators, and provide an analysis of the properties of the written prompts and the resulting AI generated code. Specifically, we explore **(A)** the context in which learners use Codex, **(B)** what learners are asking from Codex in terms of syntax and logic, **(C)** properties of prompt written by learners in terms of relation to task description, language, clarity, and prompt crafting patterns, **(D)** properties of the AI-generated code in terms of correctness, complexity, accuracy, and **(E)** how learners utilize AI-generated code in terms of placement, verification, and manual modifications. Furthermore, our analysis reveals four distinct coding approaches when writing code with an AI code generator: *AI Single Prompt*, where learners prompted Codex once to generate the entire solution to a task; *AI Step-by-Step*, where learners divided the problem into parts and used Codex to generate each part; *Hybrid*, where learners wrote some of the code themselves and used Codex to generate others; and *Manual* coding, where learners wrote the code themselves. The *AI Single Prompt* approach resulted in the highest correctness scores on code-authoring tasks, but the lowest correctness scores on subsequent code-modification tasks during training. Our results provide initial insight into how novice learners use AI code generators and the challenges and opportunities associated with integrating them into self-paced learning environments. We conclude with various signs of over-reliance and self-regulation, as well as opportunities for curriculum and tool development.

ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

CCS CONCEPTS

• **Human-centered computing → Interactive systems and tools**; • **Social and professional topics → Computing education.**

KEYWORDS

Large Language Models, OpenAI Codex, ChatGPT, Copilot, qualitative analysis, introductory programming, self-paced learning

ACM Reference Format:
Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. In *Proceedings of Koli Calling International Conference on Computing Education Research (Koli Calling).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Large Language Models (LLMs) trained on code like OpenAI Codex [12] are capable of generating functioning programs from natural language descriptions. Since publicly made available by companies like OpenAI through user-facing tools such as ChatGPT (a Q&A chatbot) or Github Copilot (an IDE-based AI coding assistant), the code generation capabilities of LLMs are becoming more accessible to a wider array of people. These tools have the potential of scaling up computing education in self-paced learning environments and broadening participation in computing by assisting beginners with debugging, code generation, code explanation, and responding to questions about code.

Despite their potential benefits, LLMs present challenges in educational contexts. Their usage could result in learner dependency, hindering code authorship without assistance. Novice coders may also struggle with technical jargon, expressing coding intent, and comprehending or verifying AI-generated code. Additionally, from an educator's perspective, issues around academic integrity and plagiarism pose valid concerns [5].

Koli Calling Conference in Computing Education Research
Koli, Finland, Nov 2023

Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming

Majeed Kazemitabaar
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
majeed@dgp.toronto.edu

Justin Chow
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
justinchow@dgp.toronto.edu

Carl Ka To Ma
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
cma@dgp.toronto.edu

Barbara J. Ericson
School of Information, University of Michigan
Ann Arbor, Michigan, USA
barbarer@umich.edu

David Weintrop
College of Education, University of Maryland
College Park, Maryland, USA
weintrop@umd.edu

Tovi Grossman
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
tovi@dgp.toronto.edu

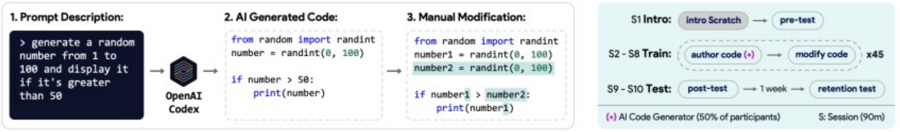


Figure 1: Left) Generate-modify usages with AI code generators. Right) Summary of our controlled study over 10 sessions.

ABSTRACT

AI code generators like OpenAI Codex have the potential to assist novice programmers by generating code from natural language descriptions, however, over-reliance might negatively impact learning and retention. To explore the implications that AI code generators have on introductory programming, we conducted a controlled experiment with 69 novices (ages 10-17). Learners worked on 45 Python code-authoring tasks, for which half of the learners had access to Codex, each followed by a code-modification task. Our results show that using Codex significantly increased code-authoring performance (1.15x increased completion rate and 1.8x higher scores) while not decreasing performance on manual code-modification tasks. Additionally, learners with access to Codex during the training phase performed slightly better on the evaluation post-tests conducted one week later, although this difference did not reach statistical significance. Of interest, learners with higher Scratch pre-test scores performed significantly better on retention post-tests, if they had prior access to Codex.

CHI '23, April 23–28, 2023, Hamburg, Germany

ACM ISBN 978-1-4503-9421-5/23/04...$15.00
https://doi.org/10.1145/3544548.3580919

CCS CONCEPTS

• **Human-centered computing → Interactive systems and tools**; • **Social and professional topics → Computing education**.

KEYWORDS

Large Language Models, AI Coding Assistants, AI-Assisted Pair-Programming, OpenAI Codex, GPT-3, ChatGPT, Copilot, Introductory Programming, K-12 Computer Science Education

ACM Reference Format:
Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23), April 23–28, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3544548.3580919

1 INTRODUCTION

Powered by the recent advancements in Deep Learning [88], Large Language Models that are trained on millions of lines of code, such as OpenAI Codex [14], can generate code from natural language descriptions (Figure 1, Left). In addition to enabling natural language programming, these AI coding assistants can perform numerous operations including code-to-code operations like code completion, translation, repair, and summarization, along with language-to-code operations such as code explanation and search [58, 79]. By generating code from simple sentences instead of formal and syntactically fixed specifications, these AI Coding Assistants may lower the barriers to entry into programming.

CHI Conference in Human Factors in Computing
Hamburg, Germany, May 2023

How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment

Majeed Kazemitabaar
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
majeed@dgp.toronto.edu

Xinying Hou
School of Information, University of Michigan
Ann Arbor, Michigan, USA
xyhou@umich.edu

Austin Henley
Microsoft Research
Redmond, Washington, USA
austinhenley@microsoft.com

Barbara J. Ericson
School of Information, University of Michigan
Ann Arbor, Michigan, USA
barbarer@umich.edu

David Weintrop
College of Education, University of Maryland
College Park, Maryland, USA
weintrop@umd.edu

Tovi Grossman
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
tovi@dgp.toronto.edu

ABSTRACT

As Large Language Models (LLMs) gain in popularity, it is important to understand how novice programmers use them and the effect they have on learning to code. We present the results of a thematic analysis on a data set from 33 learners, aged 10-17, as they independently learned Python by working on 45 code-authoring tasks with access to an AI Code Generator based on OpenAI Codex. We explore several important questions related to how learners used LLM-based AI code generators, and provide an analysis of the properties of the written prompts and the resulting AI generated code. Specifically, we explore **(A)** the context in which learners use Codex, **(B)** what learners are asking from Codex in terms of syntax and logic, **(C)** properties of prompt written by learners in terms of relation to task description, language, clarity, and prompt crafting patterns, **(D)** properties of the AI-generated code in terms of correctness, complexity, accuracy, and **(E)** how learners utilize AI-generated code in terms of placement, verification, and manual modifications. Furthermore, our analysis reveals four distinct coding approaches when writing code with an AI code generator: *AI Single Prompt*, where learners prompted Codex once to generate the entire solution to a task; *AI Step-by-Step*, where learners divided the problem into parts and used Codex to generate each part; *Hybrid*, where learners wrote some of the code themselves and used Codex to generate others; and *Manual* coding, where learners wrote the code themselves. The *AI Single Prompt* approach resulted in the highest correctness scores on code-authoring tasks, but the lowest correctness scores on subsequent code-modification tasks during training. Our results provide initial insight into how novice learners use AI code generators and the challenges and opportunities associated with integrating them into self-paced learning environments. We conclude with various signs of over-reliance and self-regulation, as well as opportunities for curriculum and tool development.

Koli Calling, 2023, Finland

ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

CCS CONCEPTS

• **Human-centered computing → Interactive systems and tools**; • **Social and professional topics → Computing education**.

KEYWORDS

Large Language Models, OpenAI Codex, ChatGPT, Copilot, qualitative analysis, introductory programming, self-paced learning

ACM Reference Format:
Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. In *Proceedings of Koli Calling International Conference on Computing Education Research (Koli Calling).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Large Language Models (LLMs) trained on code like OpenAI Codex [12] are capable of generating functioning programs from natural language descriptions. Since publicly made available by companies like OpenAI through user-facing tools such as ChatGPT (a Q&A chatbot) or Github Copilot (an IDE-based AI coding assistant), the code generation capabilities of LLMs are becoming more accessible to a wider array of people. These tools have the potential of scaling up computing education in self-paced learning environments and broadening participation in computing by assisting beginners with debugging, code generation, code explanation, and responding to questions about code.

Despite their potential benefits, LLMs present challenges in educational contexts. Their usage could result in learner dependency, hindering code authorship without assistance. Novice coders may also struggle with technical jargon, expressing coding intent, and comprehending or verifying AI-generated code. Additionally, from an educator's perspective, issues around academic integrity and plagiarism pose valid concerns [5].

Koli Calling Conference in Computing Education Research
Koli, Finland, Nov 2023

Intro: Natural Language Programming

Write a program that repeatedly generates a random number between 0 and 100 until the random number that it generates becomes equal to 50 (and then stop). Then display the number of attempts it took to generate the number.

python

Copy code

```python
import random

attempts = 0
while True:
    num = random.randint(0, 100)
    attempts += 1
    if num == 50:
        print("It took", attempts, "attempts to generate the number 50.")
        break
```

OpenAI ChatGPT

Released: November 2022

Intro: LLMs Trained on Code

```
#!/usr/bin/env ts-node

import { fetch } from "fetch-h2";

/
```

Github Copilot

Released: June 2021

Intro: Generative Operations

language *to* code

Natural Language → ***Code***

- Code Generation

Code → ***Natural Language***

- Explanation
- Evaluation

code *to* code

Code → ***Code***

- Completion
- Summarization
- Repair
- Translation

INTRODUCTORY PROGRAMMING

Enables Natural Language Programming

image source: **futurelearn.com**

INTRODUCTORY PROGRAMMING

Enables Natural Language Programming

```
> ask the user to enter a number
```

↓

LLM

↓

```python
num = int(input("enter a number: "))
```

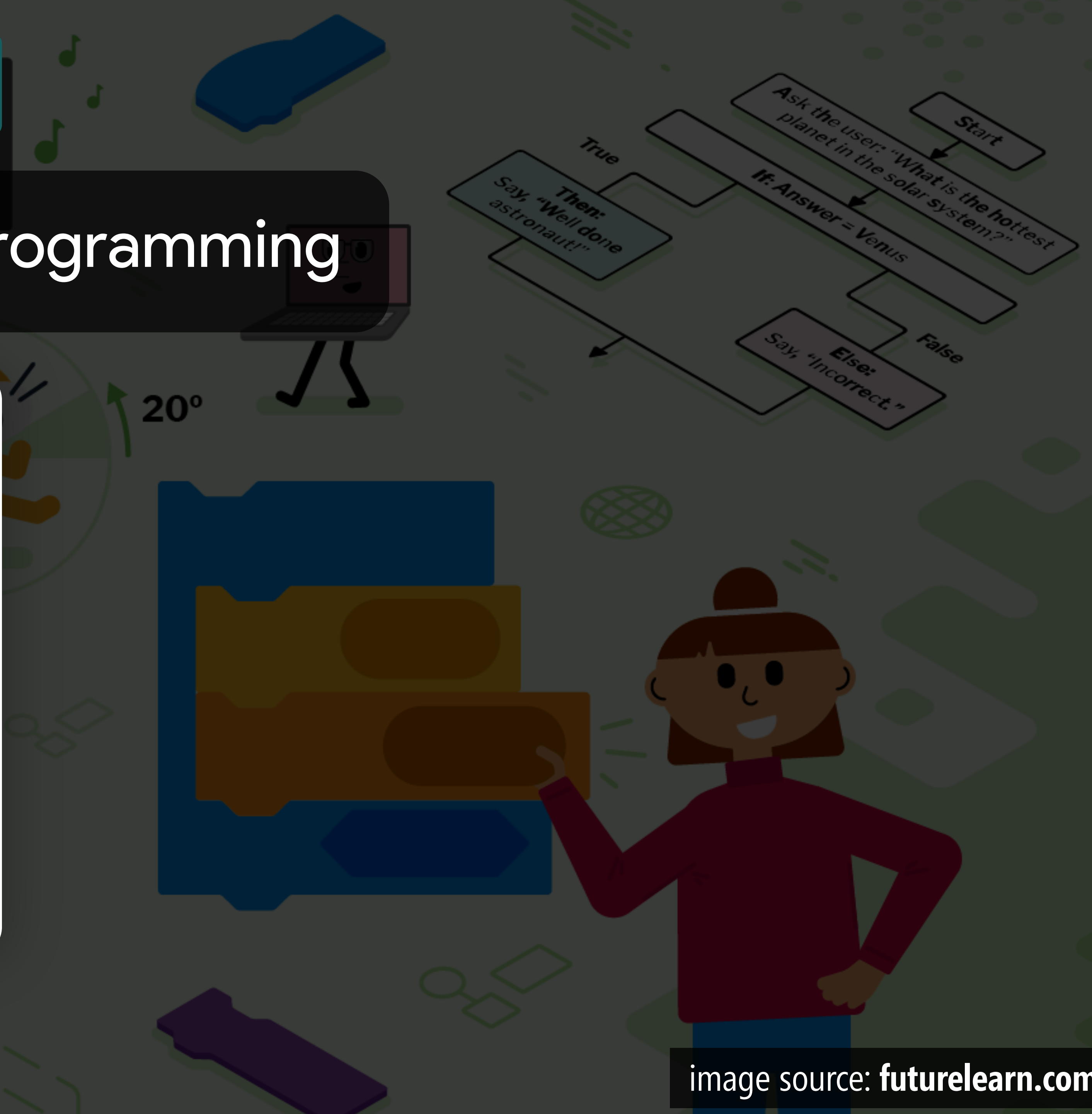


image source: futurelearn.com

Potential Benefits

Focus on **problem-solving** aspects of computing

Potential Benefits

Focus on **problem-solving** aspects of computing

Help with **debugging** and **fixing** syntax errors

Potential Benefits

- Focus on **problem-solving** aspects of computing
- Help with **debugging** and **fixing** syntax errors
- Generating a **variety** of correct solutions

Potential Benefits

- Focus on **problem-solving** aspects of computing
- Help with **debugging** and **fixing** syntax errors
- Generating a **variety** of correct solutions

Potential Drawbacks

Potential Benefits

Focus on **problem-solving** aspects of computing

Help with **debugging** and **fixing** syntax errors

Generating a **variety** of correct solutions

Potential Drawbacks

Usage Challenges:

Properly express their intentions

Potential Benefits

Focus on **problem-solving** aspects of computing

Help with **debugging** and **fixing** syntax errors

Generating a **variety** of correct solutions

Potential Drawbacks

Usage Challenges:

Properly express their intentions

Understand, verify and use AI-generated code

Intro: **Natural Language Programming**

Potential Benefits

Focus on **problem-solving** aspects of computing

Help with **debugging** and **fixing** syntax errors

Generating a **variety** of correct solutions

Potential Drawbacks

Usage Challenges:

Properly express their intentions

Understand, verify and use AI-generated code

Behavioral Challenges:

Learners might become overly-dependent

Intro: Natural Language Programming

Potential Benefits

- Focus on **problem-solving** aspects of computing
- Help with **debugging** and **fixing** syntax errors
- Generating a **variety** of correct solutions

Potential Drawbacks

Usage Challenges:

- Properly express their intentions
- Understand, verify and use AI-generated code

Behavioral Challenges:

- Learners might become overly-dependent

Ethical Issues:

- Academic integrity, plagiarism, and attribution

Intro: Impact on Learning

New York City Public School:

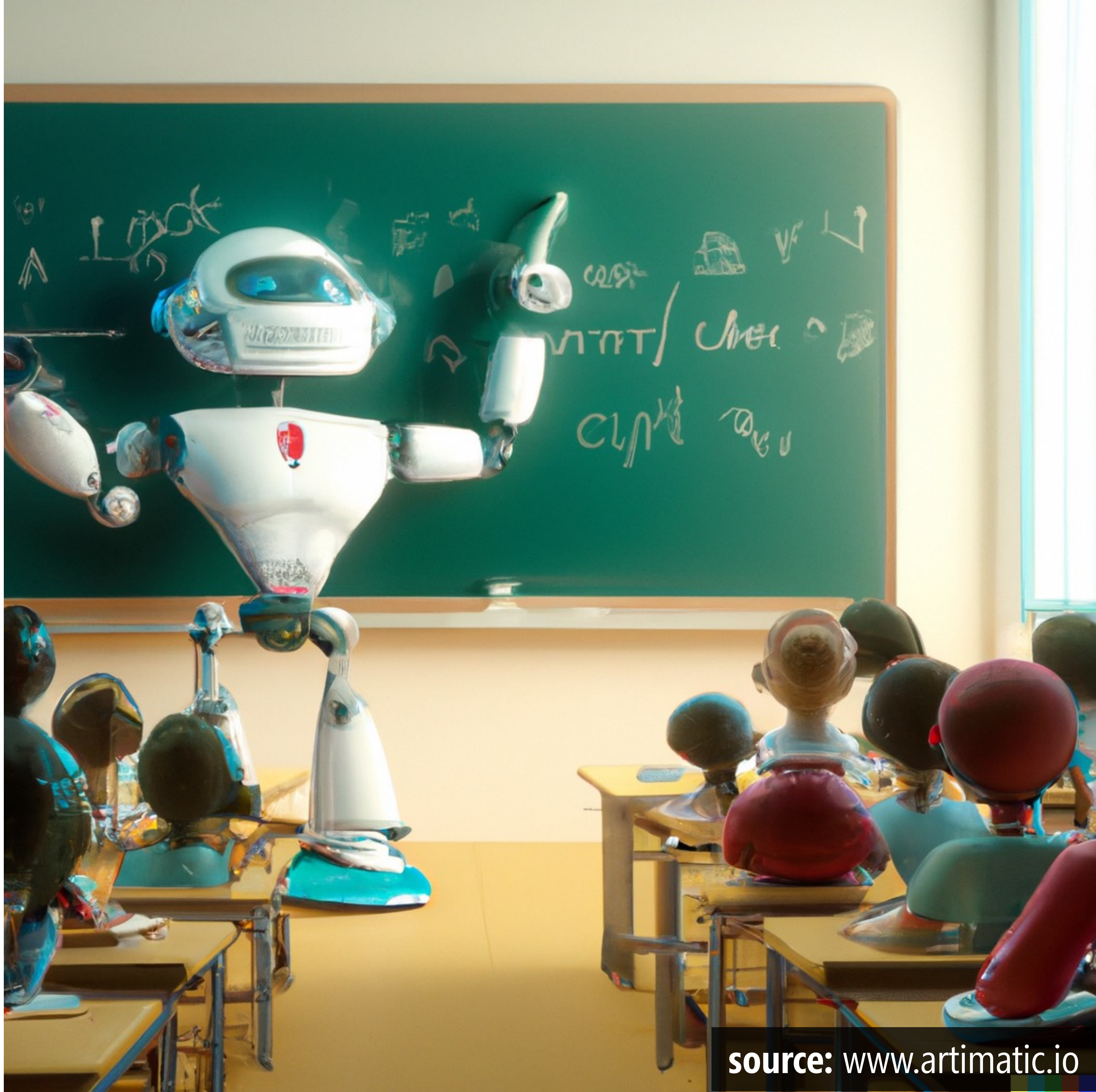
“ChatGPT doesn’t help build critical-thinking and problem-solving skills”

January 2023

Seattle public school district:

“The district does not allow cheating and requires original thought and work from students”

January 2023



RESEARCH GOAL

Scale Programming Education?

RESEARCH GOAL

Scale Programming Education?

Explore the **Impact** of using AI Code Generators on Young Students When **Learning to Write Code** for the **First Time**.

RESEARCH QUESTIONS

RESEARCH QUESTIONS

RQ1

Code Composition:

How do learners' **task performance** differ with and without AI code generators?

RESEARCH QUESTIONS

RQ1

Code Composition:

How do learners' **task performance** differ with and without AI code generators?

RQ2

Manual Code Modification:

How does prior access to the AI code generator affect learners' ability to **manually modify code**?

RESEARCH QUESTIONS

RQ1

Code Composition:

How do learners' **task performance** differ with and without AI code generators?

RQ2

Manual Code Modification:

How does prior access to the AI code generator affect learners' ability to **manually modify code**?

RQ3

Learning Retention:

What are the effects on **learning performance** and **retention** from using an AI code generator versus **not using**?

CONTROLLED STUDY

Codex Group

33 Participants

Had access to AI Code Generator

Baseline Group

36 Participants

10 Sessions

STUDY PROCEDURE

Scratch Intro + Pre-Test
1 Session

Scratch Lecture → Scratch Pre-Test

Self-Paced Python Training
7 Sessions

Code *Authoring* → Code *Modifying* x45

Evaluation
2 Sessions

Immediate Post-Test → one week later → Retention Post-Test

Session 1

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

STUDY PROCEDURE

Scratch Intro + Pre-Test
1 Session

Scratch Lecture → Scratch Pre-Test

Self-Paced Python Training
7 Sessions

Code *Authoring* → Code *Modifying* x45

Evaluation
2 Sessions

Immediate Post-Test → one week later → Retention Post-Test

Sessions 2 *to* 8

1	2	3	4	5	6	7	8	9	10

STUDY PROCEDURE

Scratch Intro + Pre-Test
1 Session

Scratch Lecture → Scratch Pre-Test

Self-Paced Python Training
7 Sessions

Code *Authoring* → Code *Modifying* x45

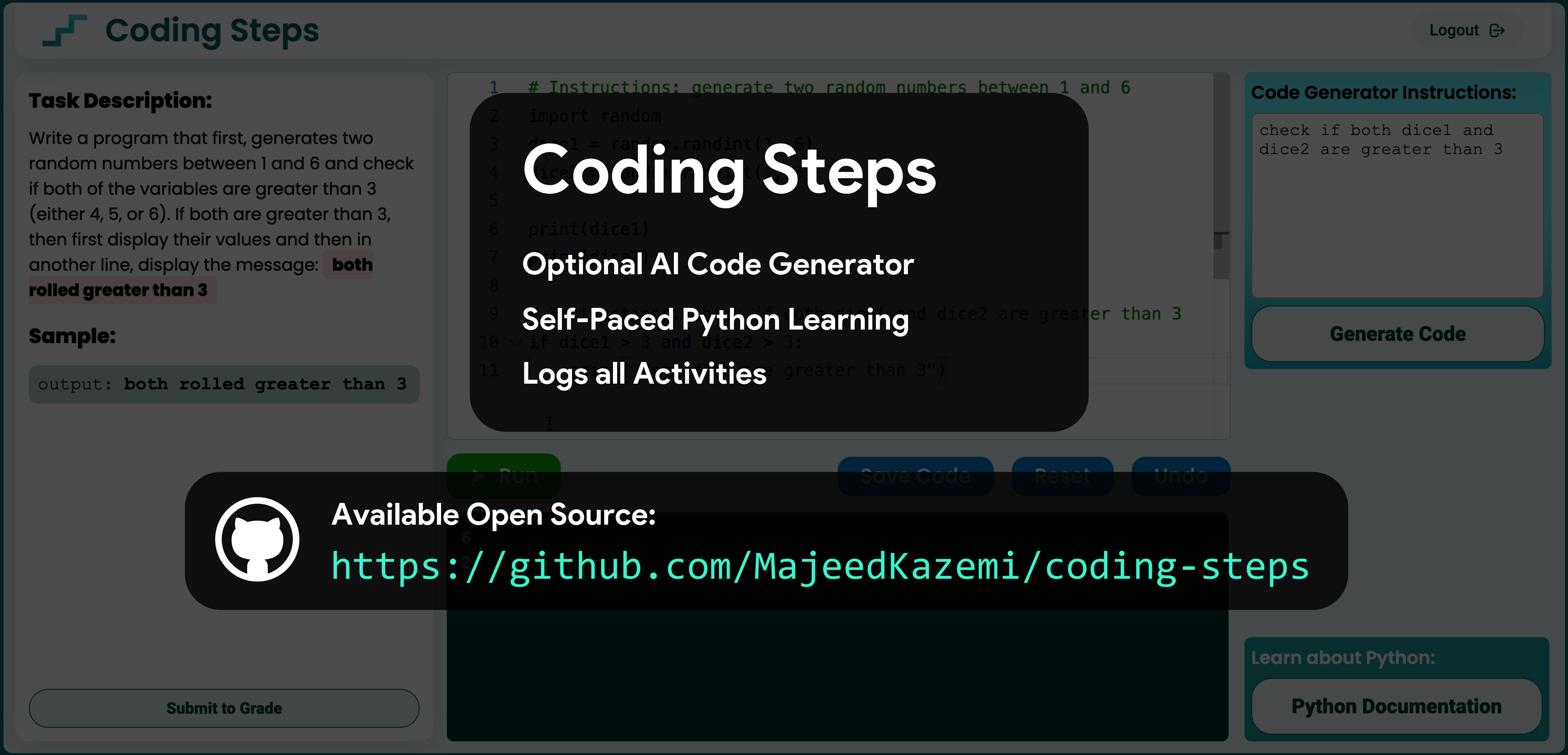
Evaluation
2 Sessions

Immediate Post-Test → one week later → Retention Post-Test

Session 9 *to* 10

1	2	3	4	5	6	7	8	9	10

AI ASSISTED PROGRAMMING



Coding Steps

Optional AI Code Generator

Self-Paced Python Learning

Logs all Activities

Available Open Source:

https://github.com/MajeedKazemi/coding-steps

AI ASSISTED PROGRAMMING

Coding Steps Demo

Coding Steps

Logout

Task Description:

Write a program that first, generates two random numbers between 1 and 6 and check if both of the variables are greater than 3 (either 4, 5, or 6). If both are greater than 3, then first display their values and then in another line, display the message: **both rolled greater than 3**

Sample:

```
output: both rolled greater than 3
```

Submit to Grade

```
1
```

▶ Run

Code Saved

Reset

Undo

Code Generator Instructions:

Describe the behavior of the code to be generated...

Generate Code

Learn about Python:

Python Documentation

User Study

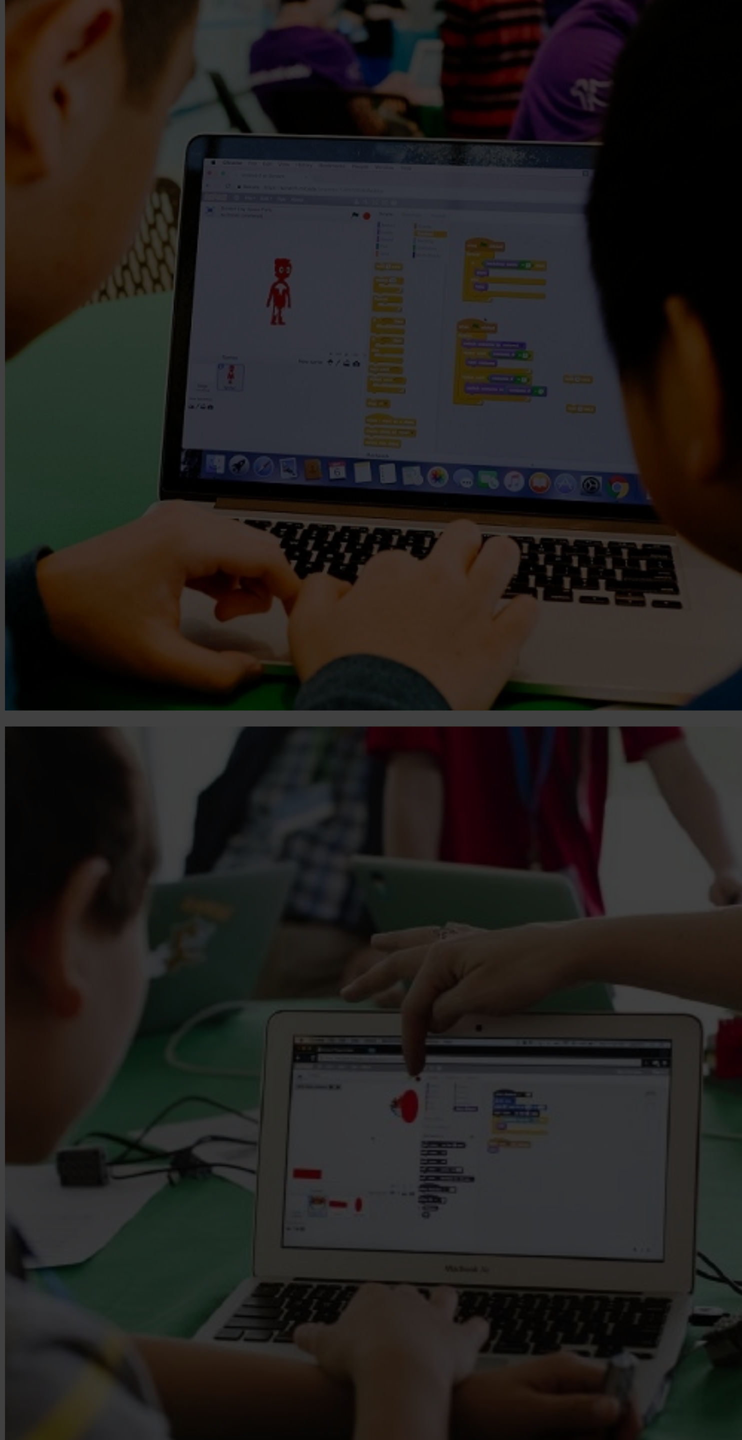
PARTICIPANTS

Total Participants: 69 (21 female, 48 male)

Ages: 10 – 17 (*M*=**12.53**, *SD*=**1.83**)

Recruitment: from multiple coding camps

Prior Programming Experience: 64 indicated using Scratch



Study Procedure

Intro + Scratch Pre-Test

1 Session

STUDY PROCEDURE

Intro + Scratch Pre-Test

1. Scratch Lecture (75 mins)

Topics: variables, operators, conditionals, loops, and arrays

2. Scratch Pre-Test (45 mins)

25 Multiple-Choice Questions

Same Topics

Intro + Pre-Test

Scratch Lecture → Scratch Pre-Test

Self-Paced Python Training

Code *Authoring* → Code *Modifying* x45

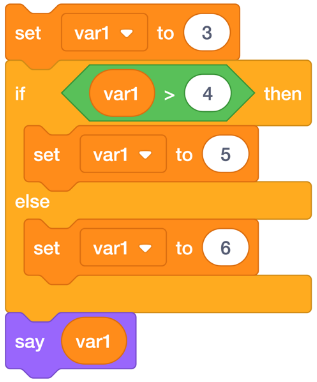
Evaluation

Immediate Post-Test → one week later → Retention Post-Test

STUDY PROCEDURE

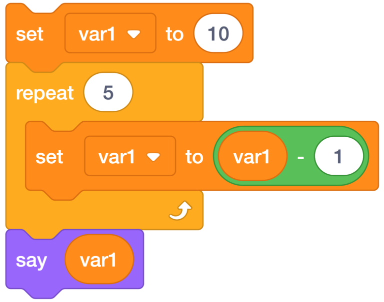
Scratch Pre-Test Samples

What does this program say?



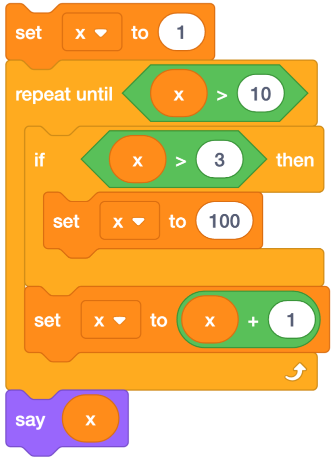
○ 3 ○ 4 ○ 5 ○ 6

What does this program say?

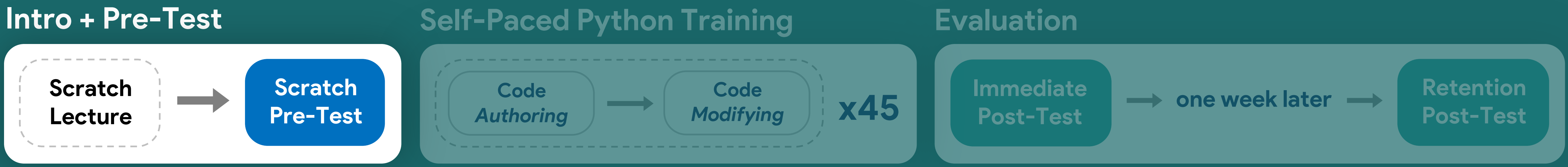


○ var1 ○ 10 ○ 5 ○ 9

What does this program say?



○ 100 ○ 10 ○ 101 ○ 11



Study Procedure

Self-Paced Python Training

7 Sessions

STUDY PROCEDURE
Two Normalized Groups

Codex Group

- **Count:** 33 Participants
- **Gender:** 11 Female

Scratch Pre-Test: 63%

Baseline Group

- **Count:** 36 Participants
- **Gender:** 10 Female

Scratch Pre-Test: 60%

Intro + Pre-Test

Scratch Lecture → Scratch Pre-Test

Self-Paced Python Training

Code *Authoring* → Code *Modifying* **x45**

Evaluation

Immediate Post-Test → one week later → Retention Post-Test

STUDY PROCEDURE — Python Topics

basics
8 coding + 6 MCQ

data-types
4 coding + 4 MCQ

conditionals
8 coding + 10 MCQ

loops
18 coding + 9 MCQ

arrays
7 coding + 10 MCQ

Intro + Pre-Test

Scratch Lecture → Scratch Pre-Test

Self-Paced Python Training

Code *Authoring* → Code *Modifying* **x45**

Evaluation

Immediate Post-Test → one week later → Retention Post-Test

STUDY PROCEDURE

Authoring + Modifying Tasks

1. Authoring Task

Task Description:

```
Repeatedly generate a random number from 0 to
100 until it generates 50. Then display the
number of times it took to generate the number.
```

Sample Output:

```
It took 27 attempts.
```

Codex Group:
Access to **AI Code Generation**

2. Modifying Task

Task Description:

```
Modify the program so it stops on any of the
numbers 25, 50, or 75.
```

```python
from random import randint
num = randint(0, 100)
count = 0
while num != 50:
    num = randint(0, 100)
    count += 1
print(str(count) + " attempts.")
```

Without AI Code Generation
(Regardless of Condition)

Intro + Pre-Test

Scratch Lecture → Scratch Pre-Test

Self-Paced Python Training

Code *Authoring* → Code *Modifying* x45

Evaluation

Immediate Post-Test → one week later → Retention Post-Test

Study Procedure

Evaluation Post-Tests

2 Sessions

STUDY PROCEDURE **Evaluation Post-Test**

1. Immediate Post-Test

- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

one week later →

2. Retention Post-Test

- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

No Python Documentation * No Instructor Hints * No AI Code Generators

Intro + Pre-Test

Scratch Lecture → Scratch Pre-Test

Self-Paced Python Training

Code *Authoring* → Code *Modifying* x45

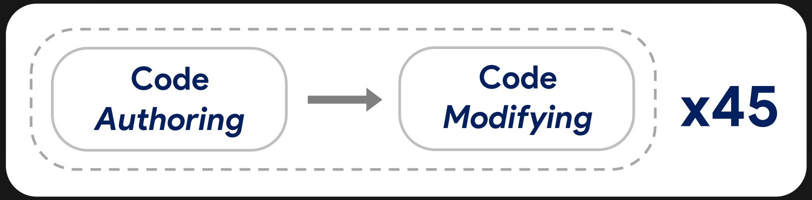
Evaluation

Immediate Post-Test → one week later → Retention Post-Test

Results

Self-Paced Python Training

7 Sessions



Differences in task performance measures

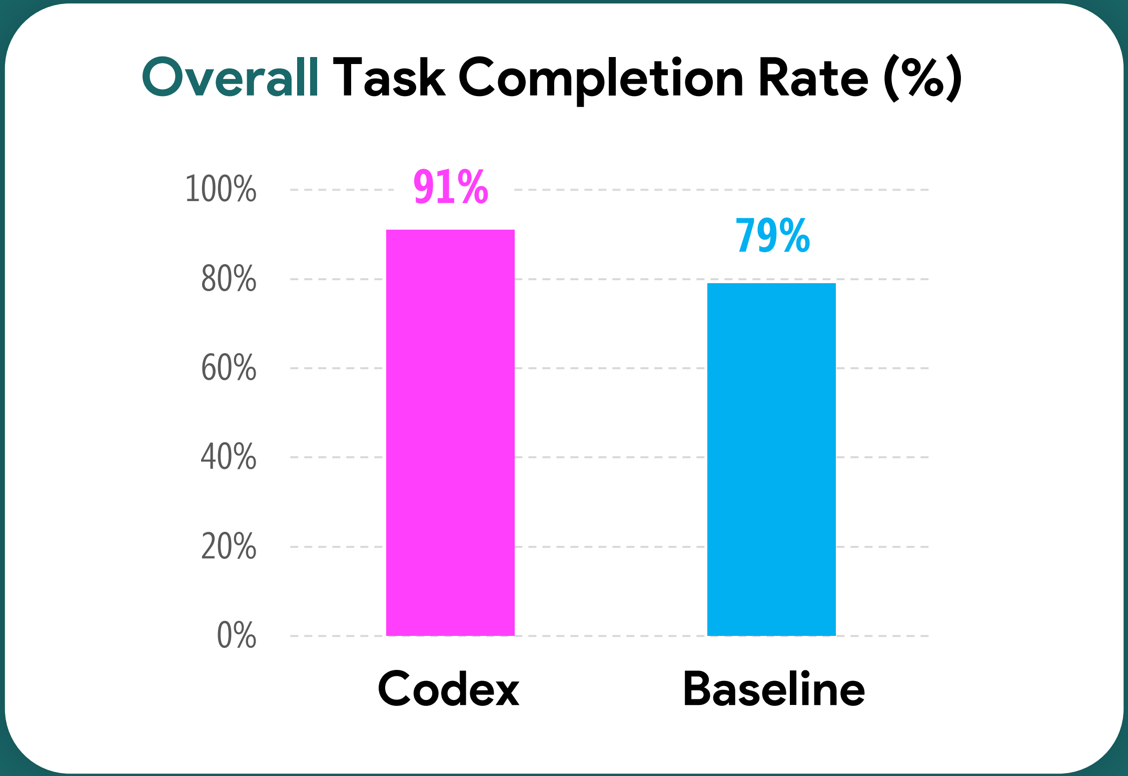
Overall Completion rate (progress)

Task Completion time

Task Correctness score

RESULTS Training Phase

Authoring + Modifying Tasks

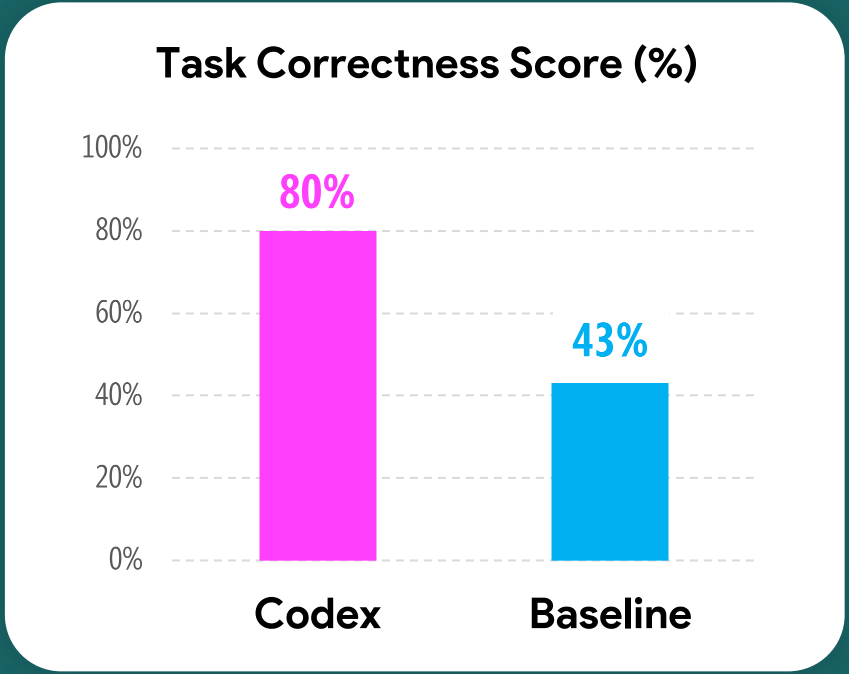


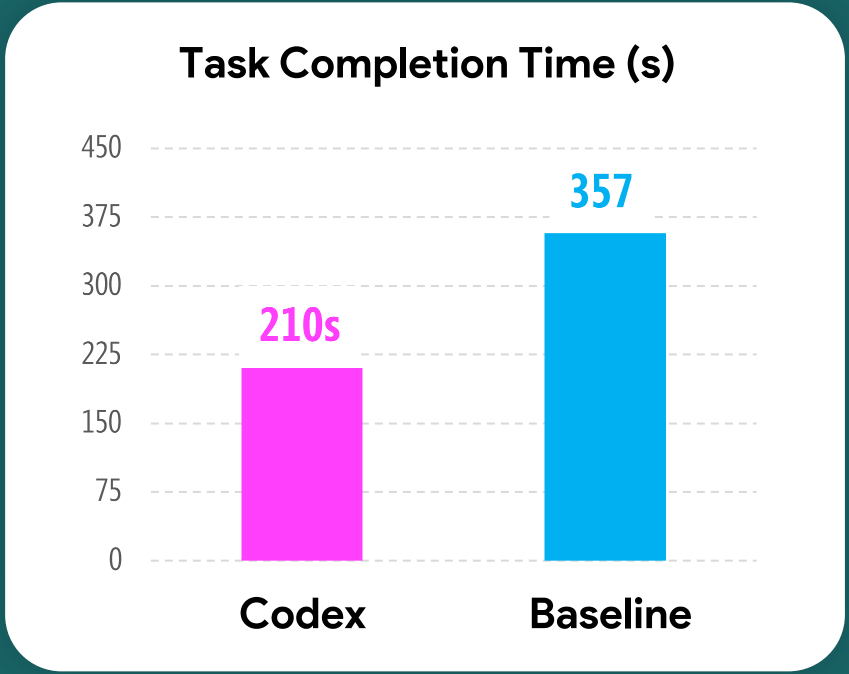
Significant Difference: *p*<.006

RESULTS

Training Phase

Authoring Tasks





RESULTS Training Phase

AI Code Generator Usage

- Students used the code generator n=1646 times (1.21 times per task)
- 32% (n=530) of prompts were an exact copy of the task description
- Final code of 49% tasks was 100% AI generated (unmodified)

RESULTS Training Phase

AI Code Generator Usage

- Students used the code generator n=1646 times (1.21 times per task)
- 32% (n=530) of prompts were an exact copy of the task description
- Final code of 49% tasks was 100% AI generated (unmodified)

RESULTS Training Phase

AI Code Generator Usage

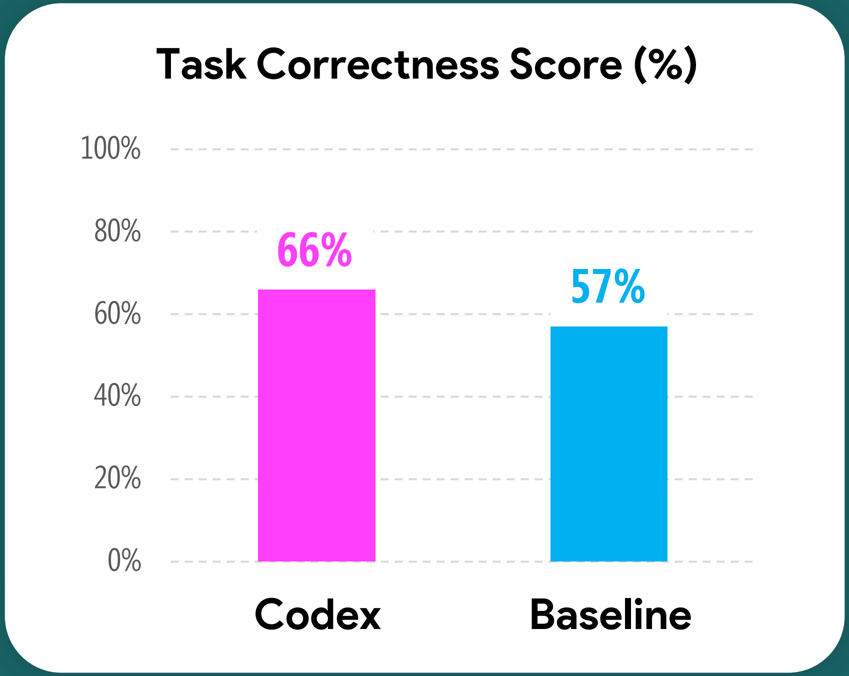
- Students used the code generator n=1646 times (1.21 times per task)
- 32% (n=530) of prompts were an exact copy of the task description
- Final code of 49% tasks was 100% AI generated (unmodified)

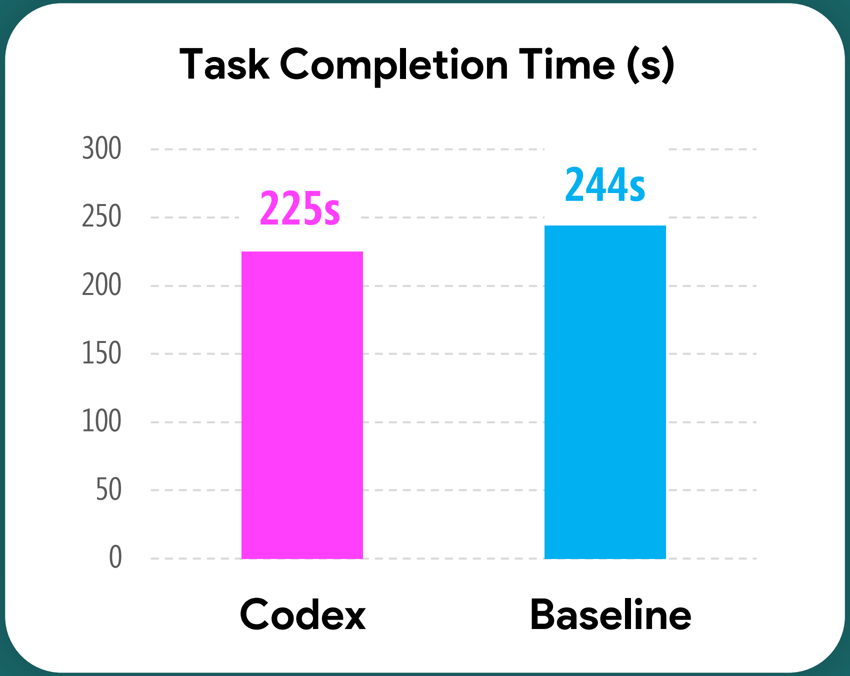
Differences in manual code modification

Without the AI Code Generator

RESULTS Training Phase

Modifying Tasks
without AI code generators





Differences in Learning Performance and Retention

RESULTS Immediate Post-Test

1. Immediate Post-Test

- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

one week later

2. Retention Post-Test

- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

No Python Documentation * No Instructor Hints * No AI Code Generators

Intro + Pre-Test: Scratch Lecture → Scratch Pre-Test

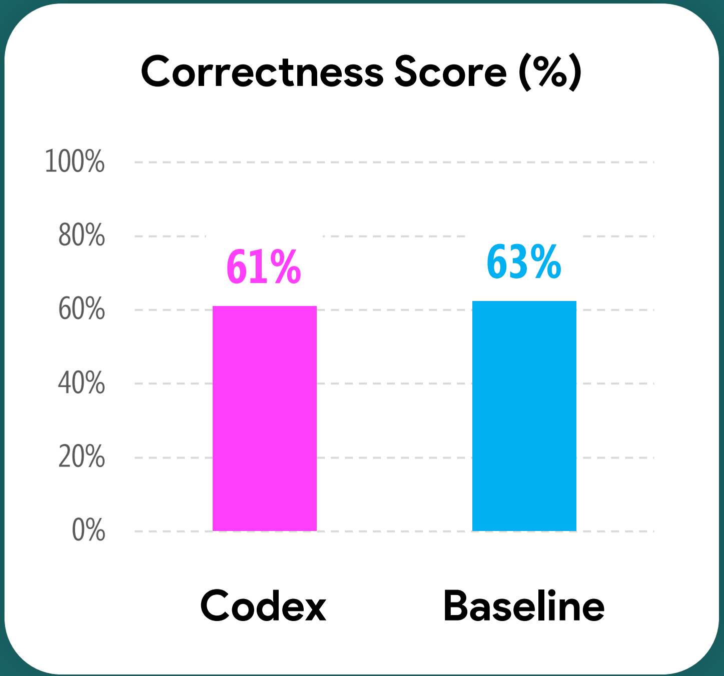
Self-Paced Python Training: Code *Authoring* → Code *Modifying* x45

Evaluation: Immediate Post-Test → one week later → Retention Post-Test

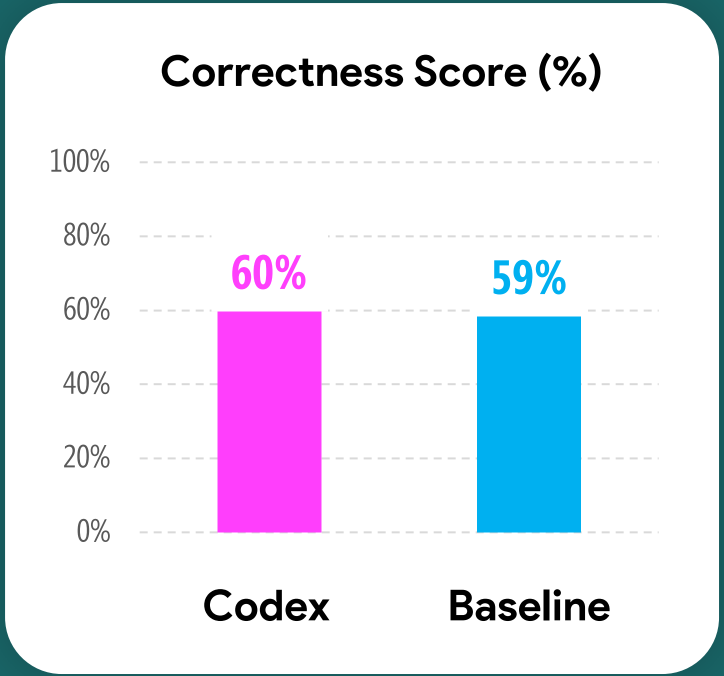
RESULTS

Immediate Post-Test

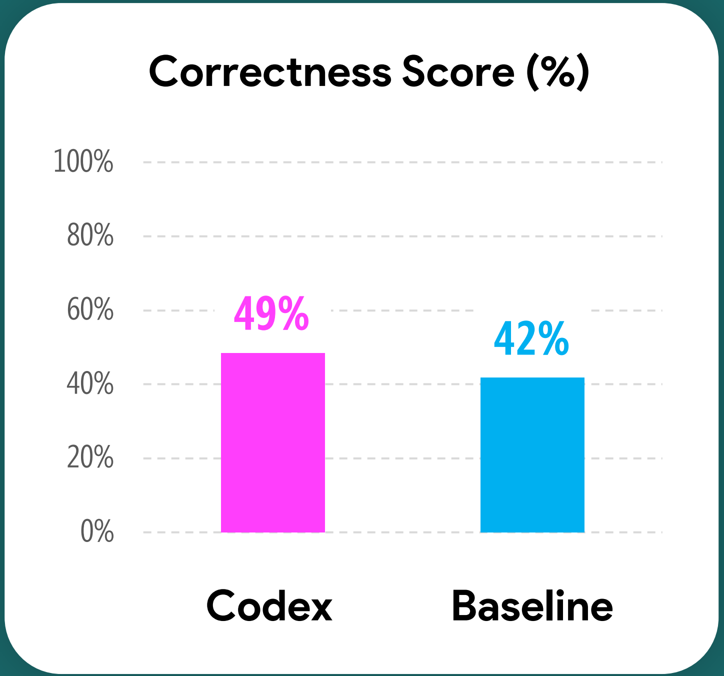
Authoring



Modifying



Multiple-Choice



RESULTS

Retention Post-Test

1. Immediate Post-Test

- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

one week later

2. Retention Post-Test

- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

No Python Documentation * No Instructor Hints * No AI Code Generators

Intro + Pre-Test

Scratch Lecture → Scratch Pre-Test

Self-Paced Python Training

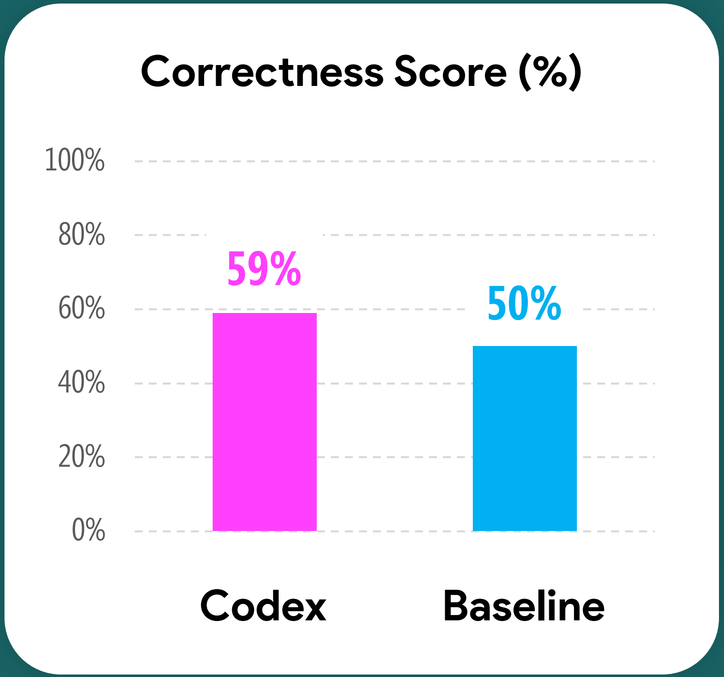
Code *Authoring* → Code *Modifying* x45

Evaluation

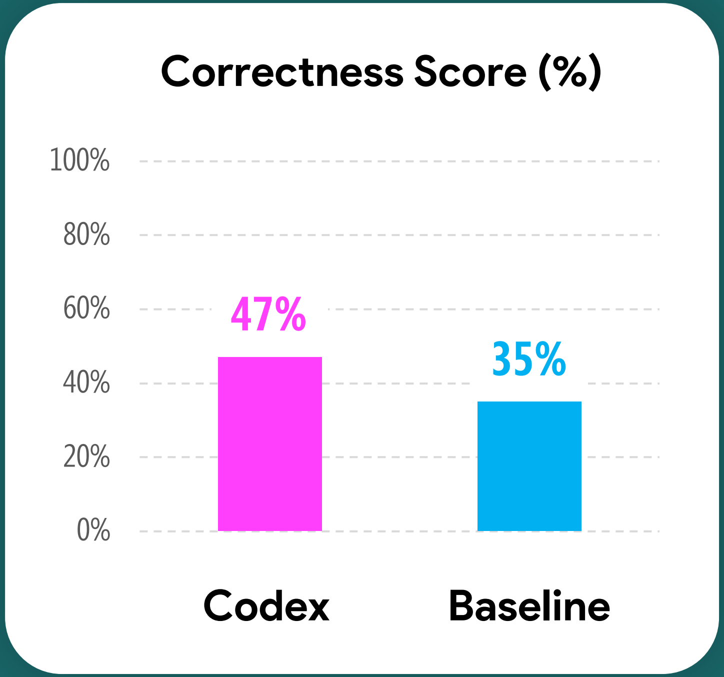
Immediate Post-Test → one week later → Retention Post-Test

RESULTS Retention Post-Test

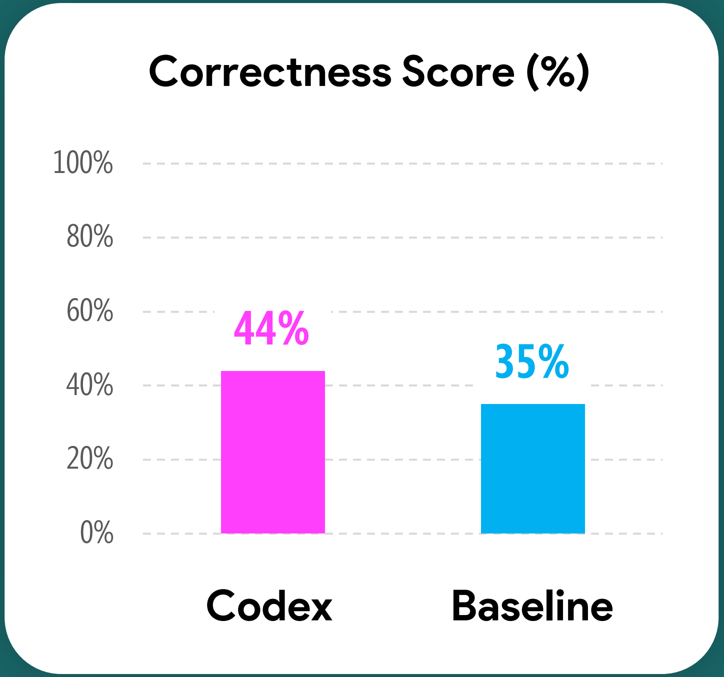
Authoring



Modifying

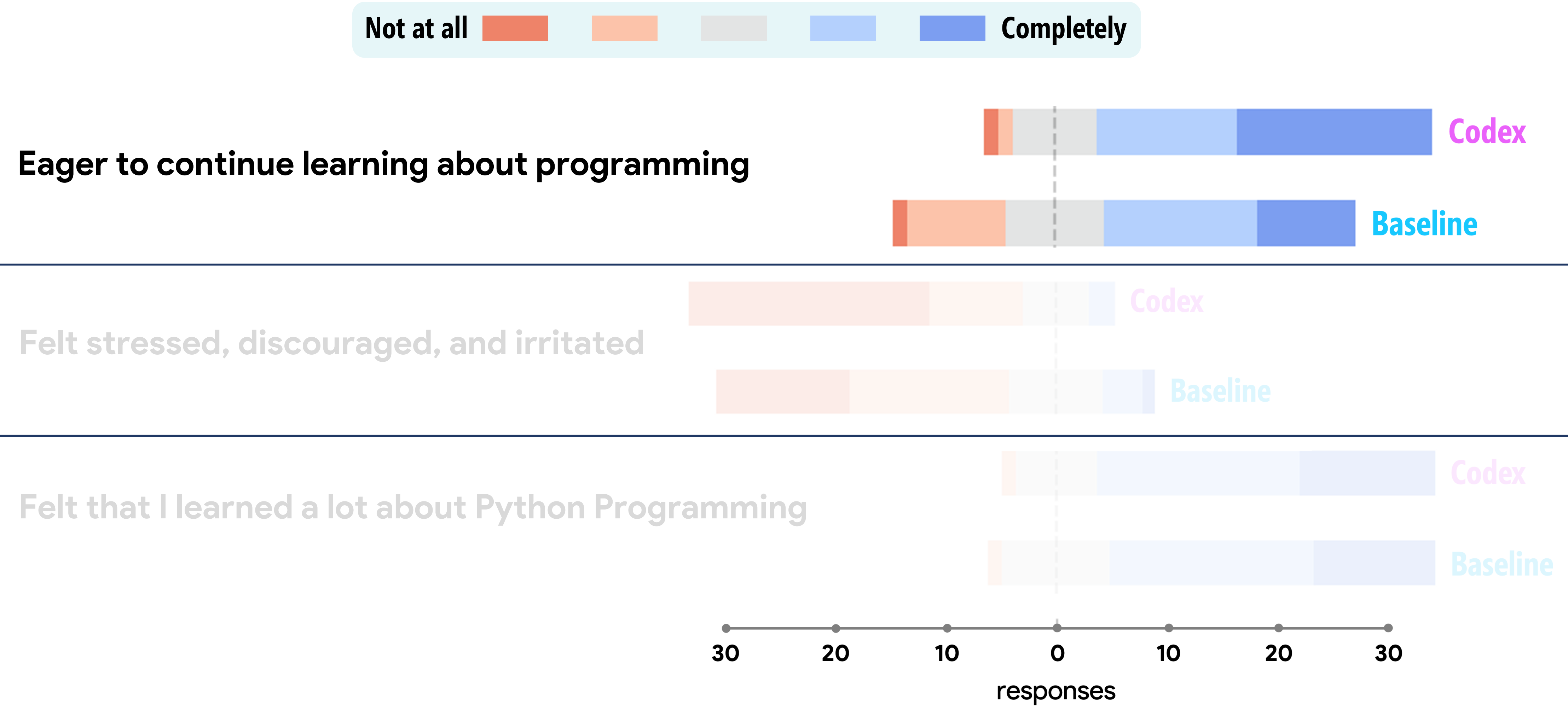


Multiple-Choice

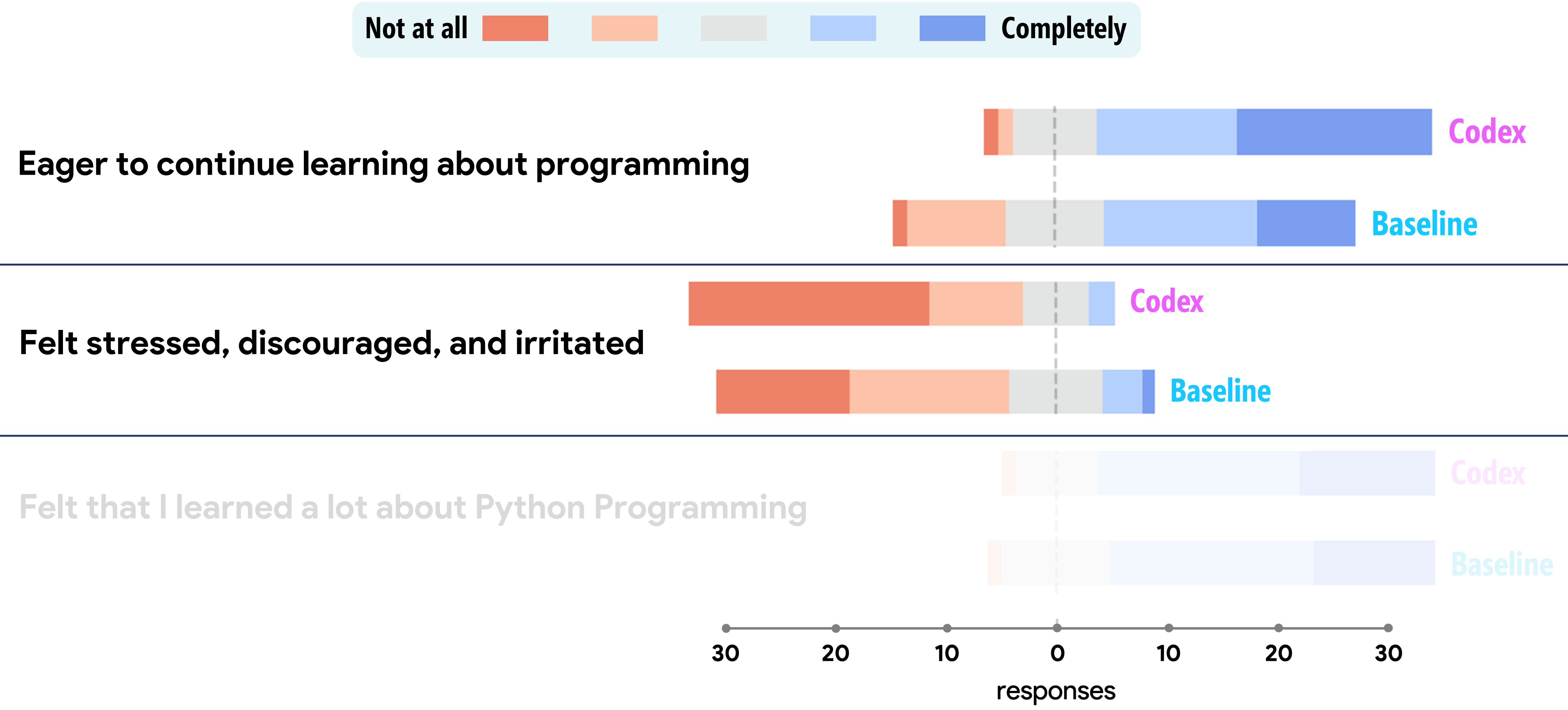


Differences in Perceptions about Learning and Frustration

RESULTS Student Perceptions

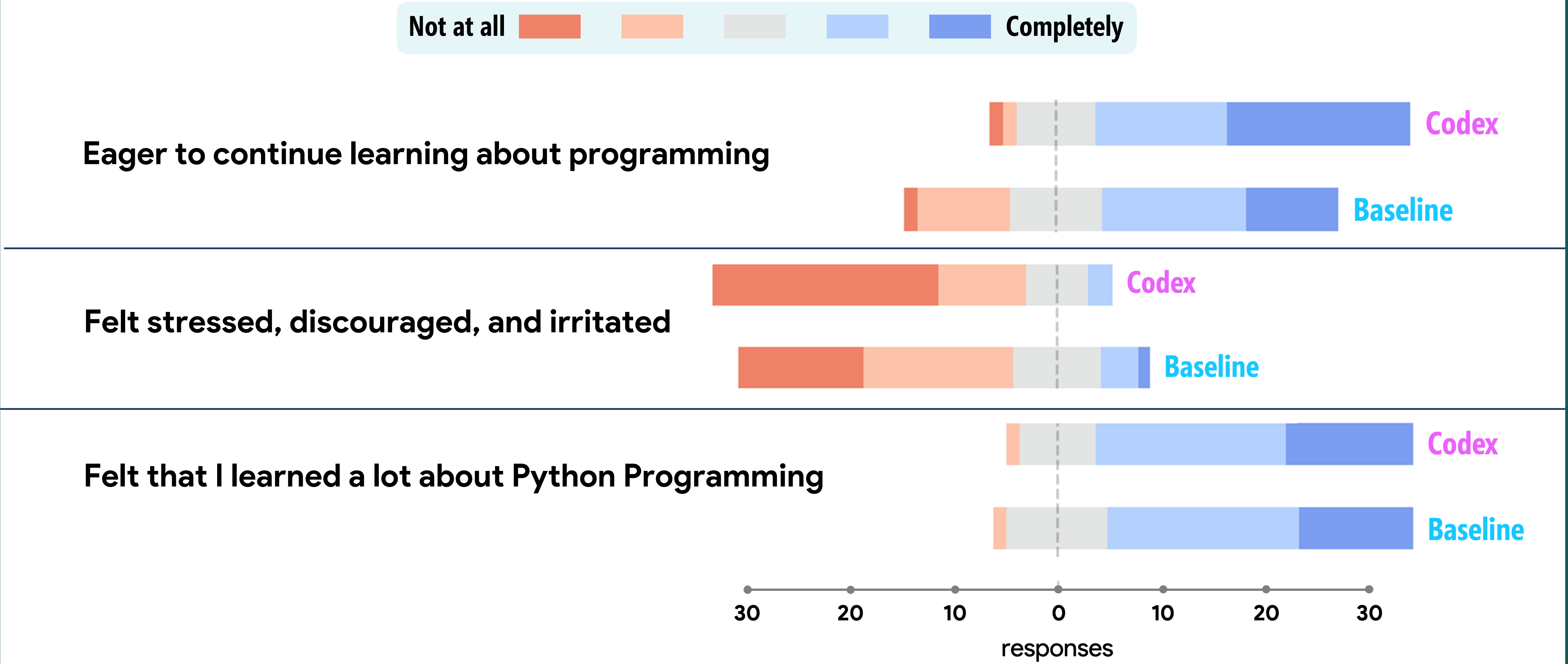


RESULTS Student Perceptions



RESULTS

Student Perceptions



KEY TAKEAWAYS

Overall, having access to AI Code Generators:

- Significantly increased completion rate of tasks
- Significantly Increased code-authoring performance (correctness)
- Did not decrease manual code modification performance
- Felt more motivated, and less stressed during the training phase
- Slightly increased performance on retention tests

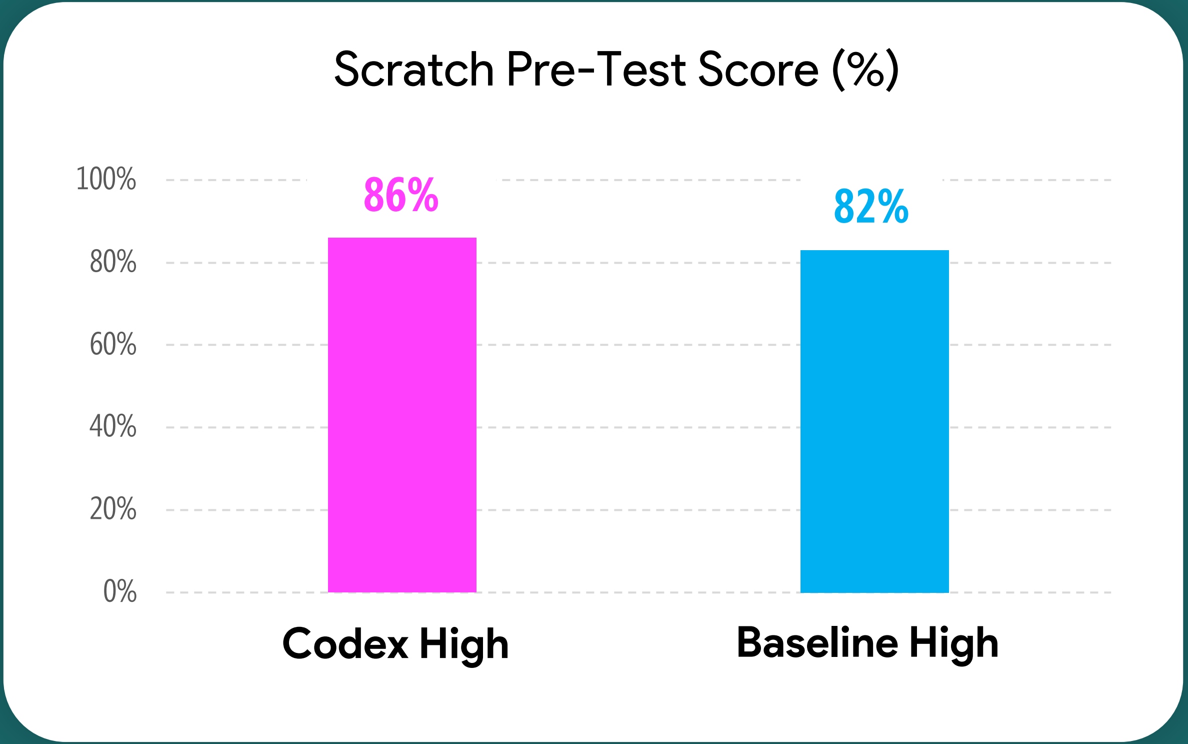
But how...?

Let's dig deeper...

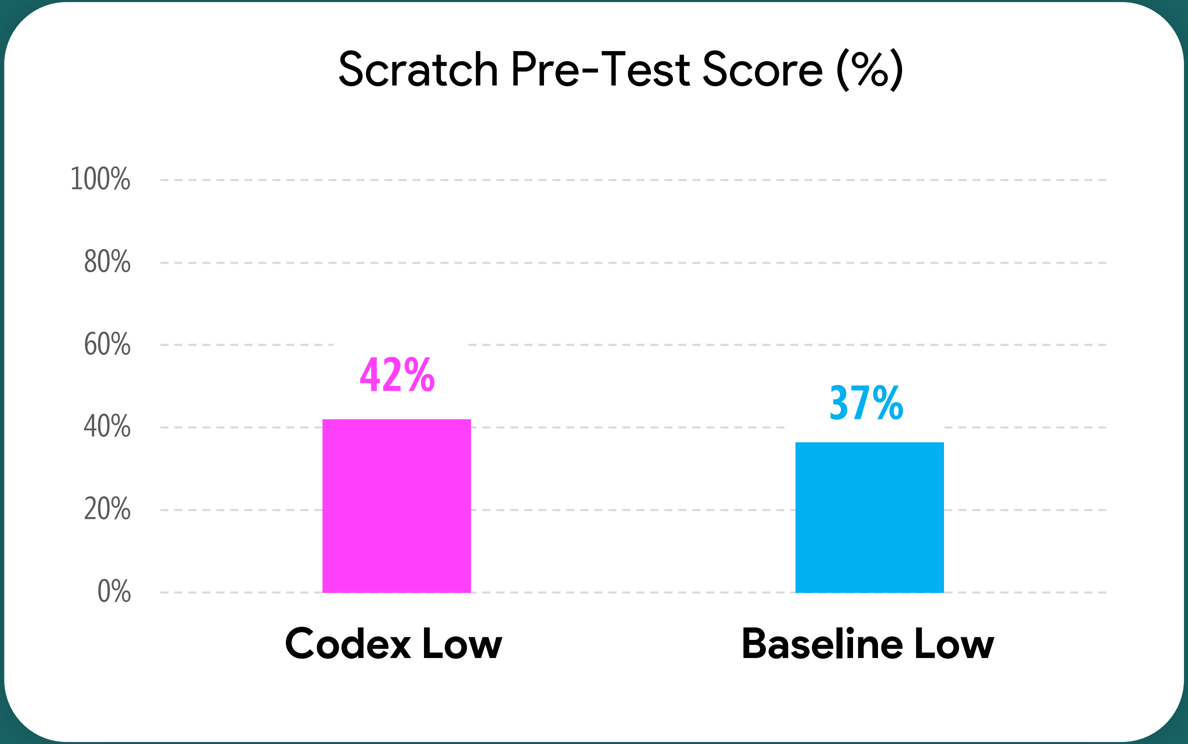
How prior programming skills affects learning performance with and without Codex?

RESULTS | **Effect of Prior Programming**

Divided learners into **four groups** based on **Scratch pre-test scores** and access to **Codex**



Top 50%



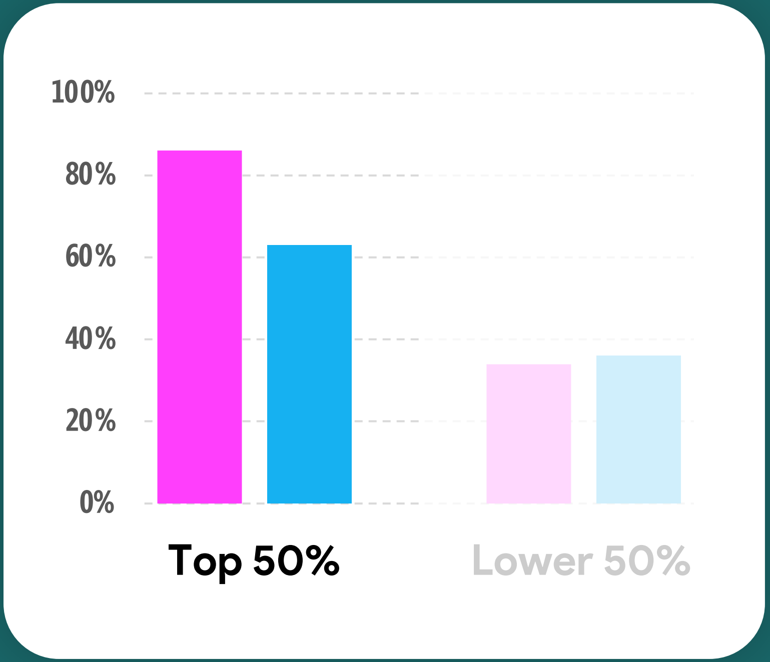
Lower 50%

RESULTS

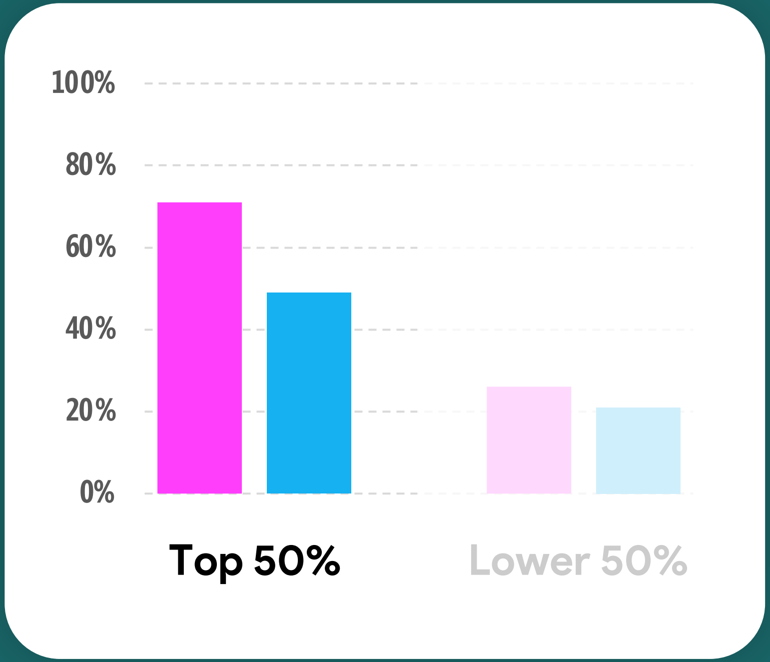
Effect of Prior Programming

Evaluation Phase: Retention Test

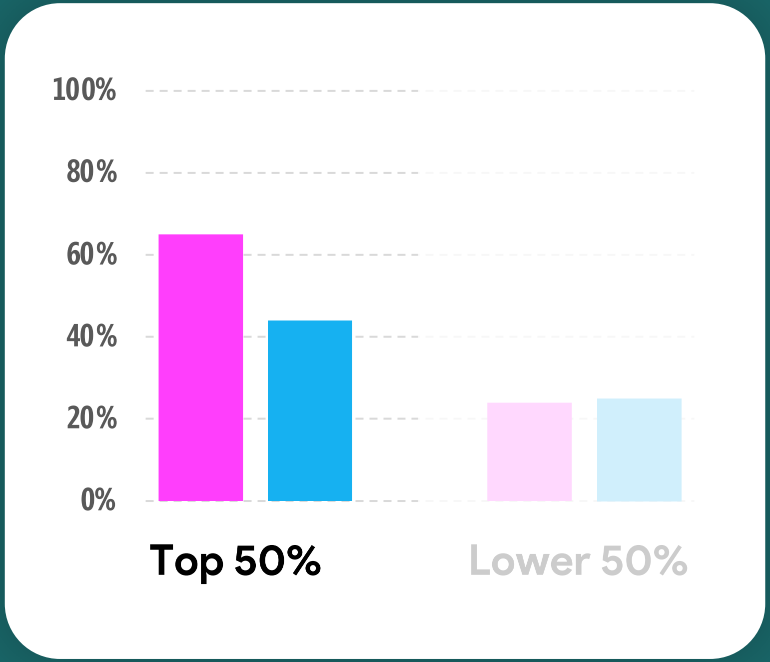
Authoring



Modifying



Multiple-Choice

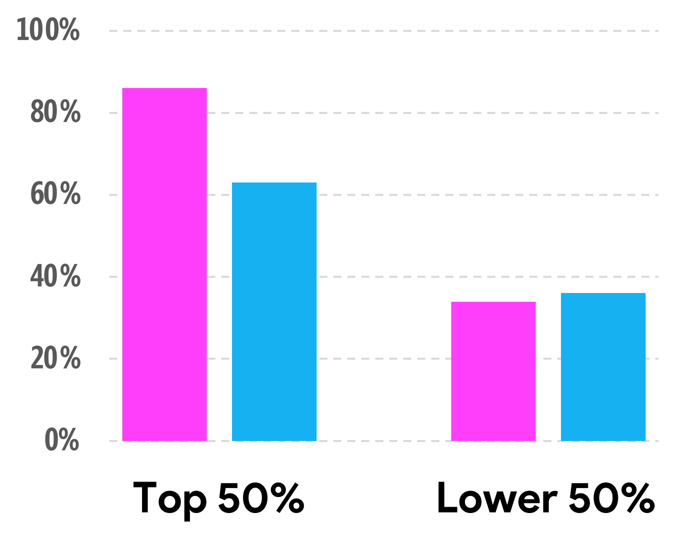


RESULTS

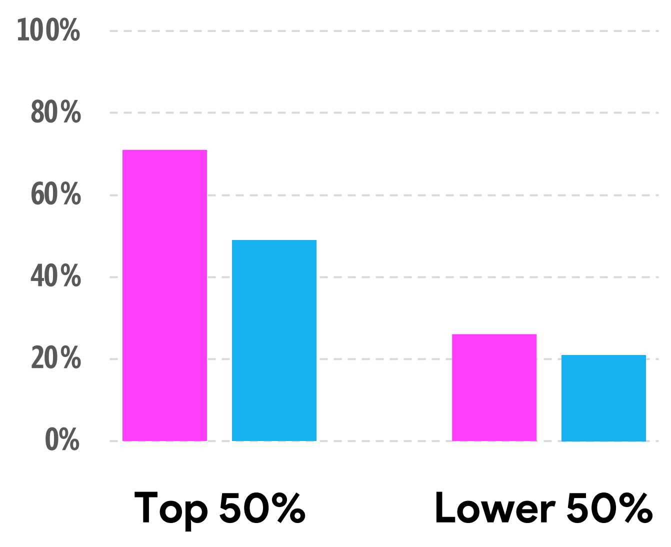
Effect of Prior Programming

Evaluation Phase: Retention Test

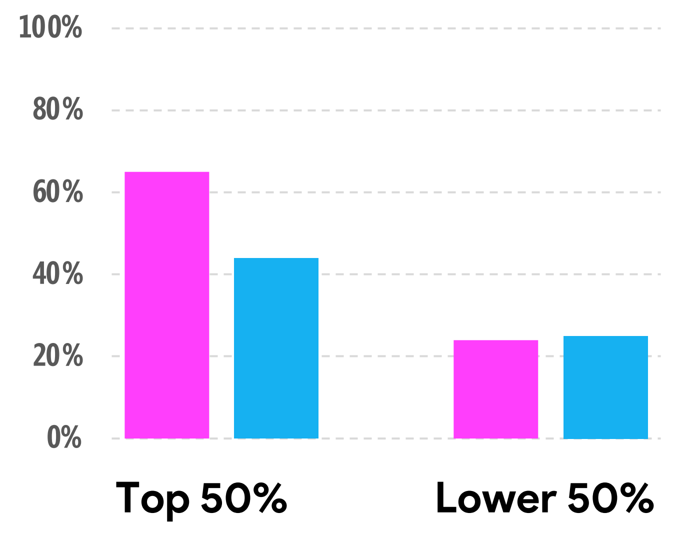
Authoring



Modifying



Multiple-Choice



Part Two:

To understand the **benefits** and **drawbacks** of LLM-powered Coding tools, it’s crucial to know ***how*** students use them

We analyzed usage patterns of students using Codex

RESEARCH QUESTIONS

RQ1 How Novices Use AI Code Generators?

RESEARCH QUESTIONS

RQ1 How Novices Use AI Code Generators?

RQ2 Effect of Coding Approaches on Learning?

Collected Data:

- 1379 submitted tasks (356 manually, without Codex)
- 1666 Codex usages (1.62 usage per task)
- Code edit logs + Console run logs + Codex usages

Analysis Interface

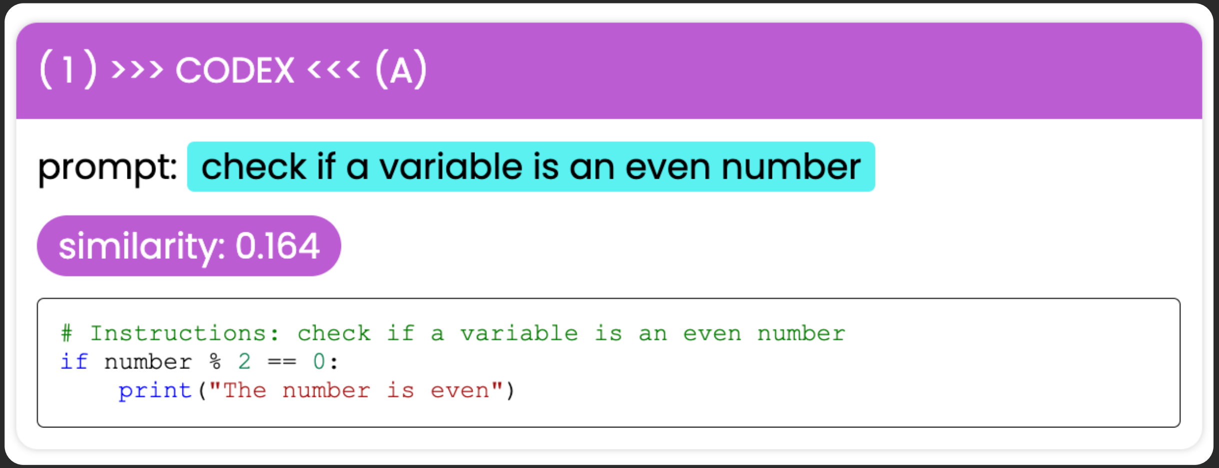
Sequence of Actions:

- Codex
- Manual Edit
- Run Code
- Manual Edit
- Codex
- Run Code
- Submit

Analysis Interface

Codex Usage

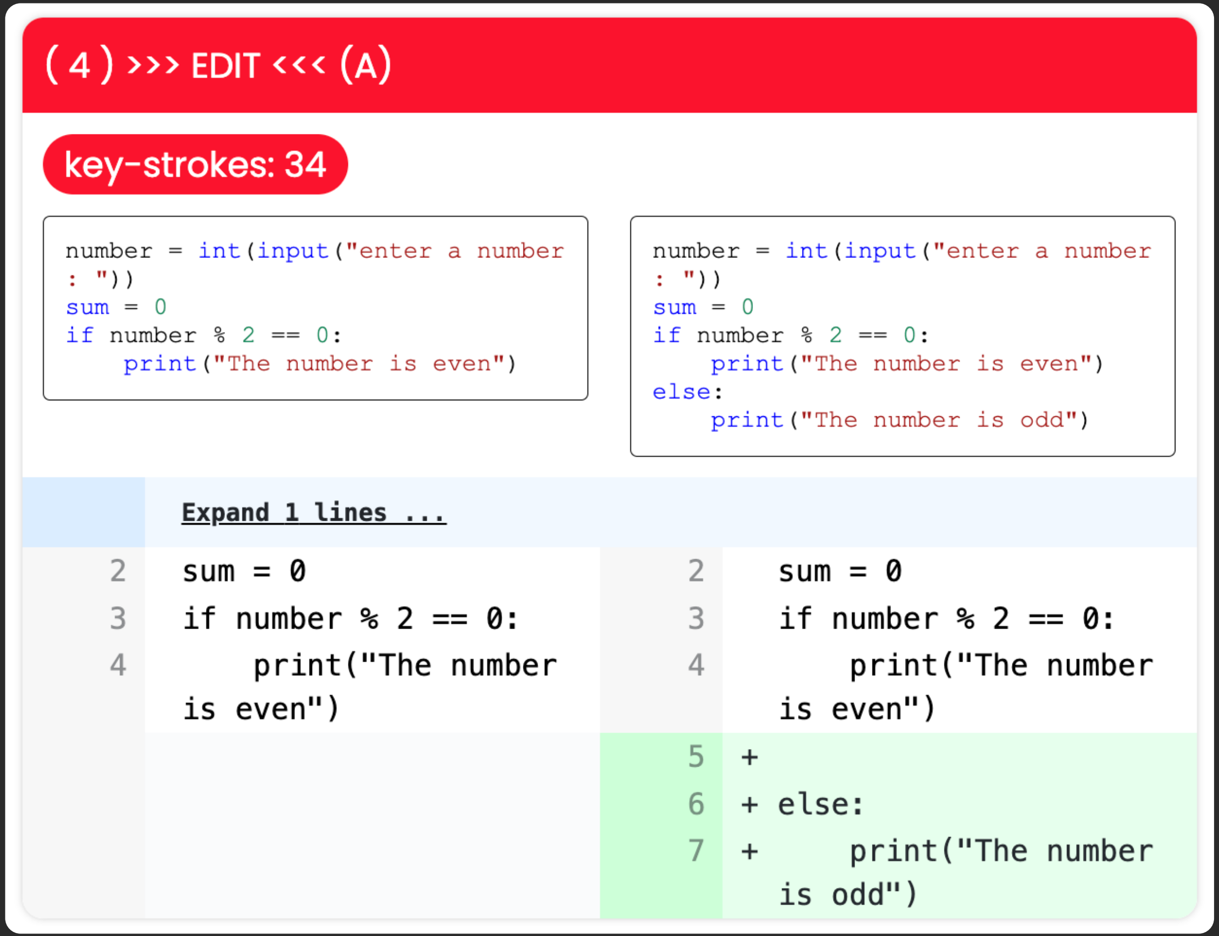
- Prompt Message
- Similarity with Task Description
- Generated Code



Analysis Interface

Manual Code Edit

- Code Before Edit vs. After Edit
- **Diff:** Before vs. After
- Key-Strokes Count



Analysis Interface

Code Execution

- Code that was Executed
- Console Input and Output

(3) >>> RUN <<< (A)

```
number = int(input("enter a number: "))
sum = 0
if number % 2 == 0:
    print("The number is even")
```

console output:

```
-->> enter a number:
<<-- 7
-->> enter a number:
<<-- 8
-->> The number is even
```

Analysis Interface

Code Submission

- Code that was Submitted
- Any feedback provided by TAs

(23) >>> SUBMIT <<< (A)

```
start = int(input("Enter a start number: "))
end = int(input("Enter an end number: "))
sum = 0
for i in range(start, end + 1):
    if i % 2 == 0:
        sum += i
print(sum)
```

Analysis Interface

Sequence of Actions:

- Codex
- Manual Edit
- Run Code
- Manual Edit
- Codex
- Run Code
- Submit

Results

When did Learners Use Codex?

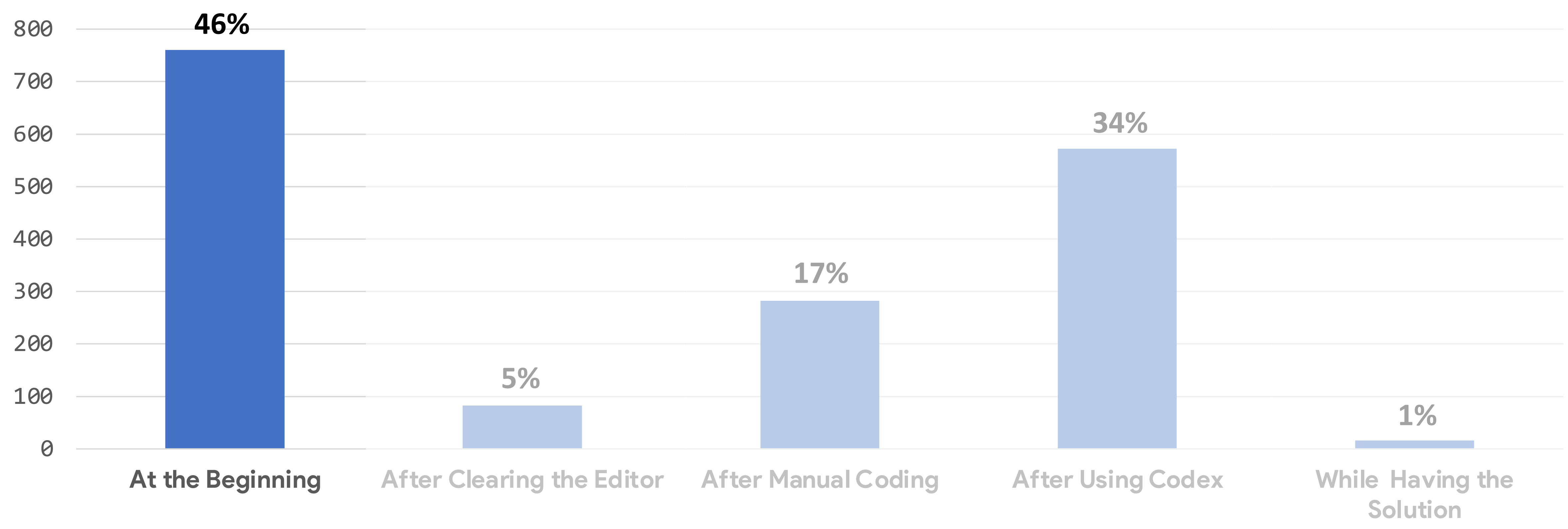
Focus of Thematic Analysis:

- Prior manual edits
- Prior codex usage
- Existing state of code

RQ1 A

When did Learners Use Codex?

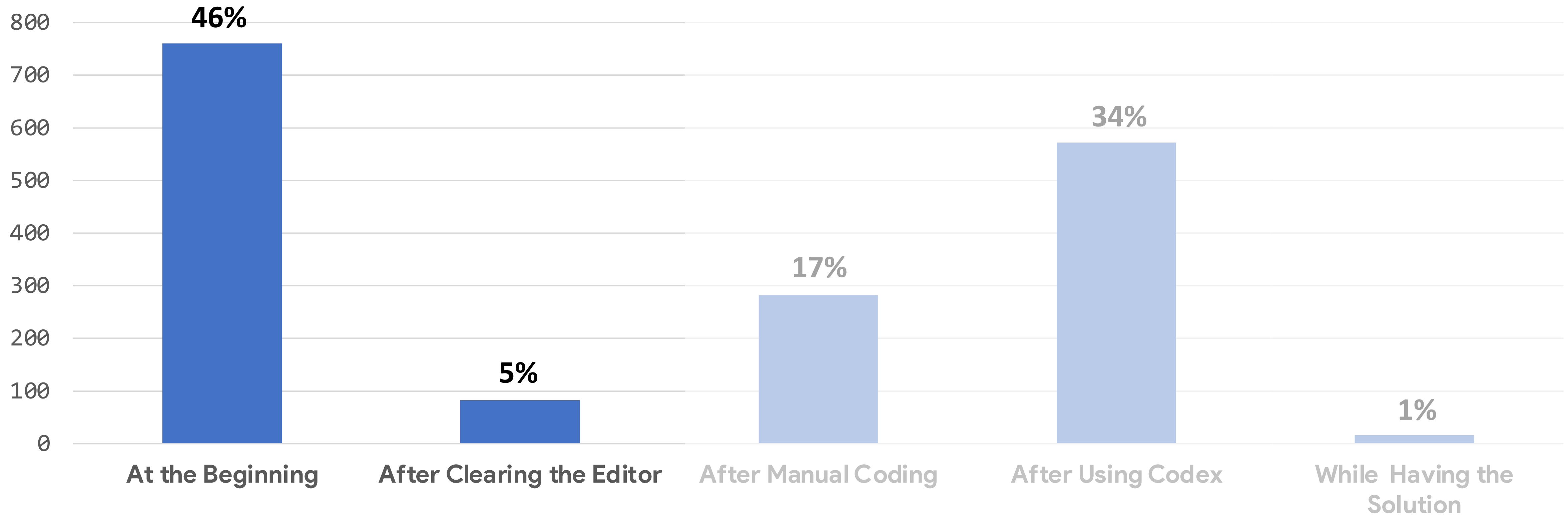
Five Primary Situations:



RQ1 A

When did Learners Use Codex?

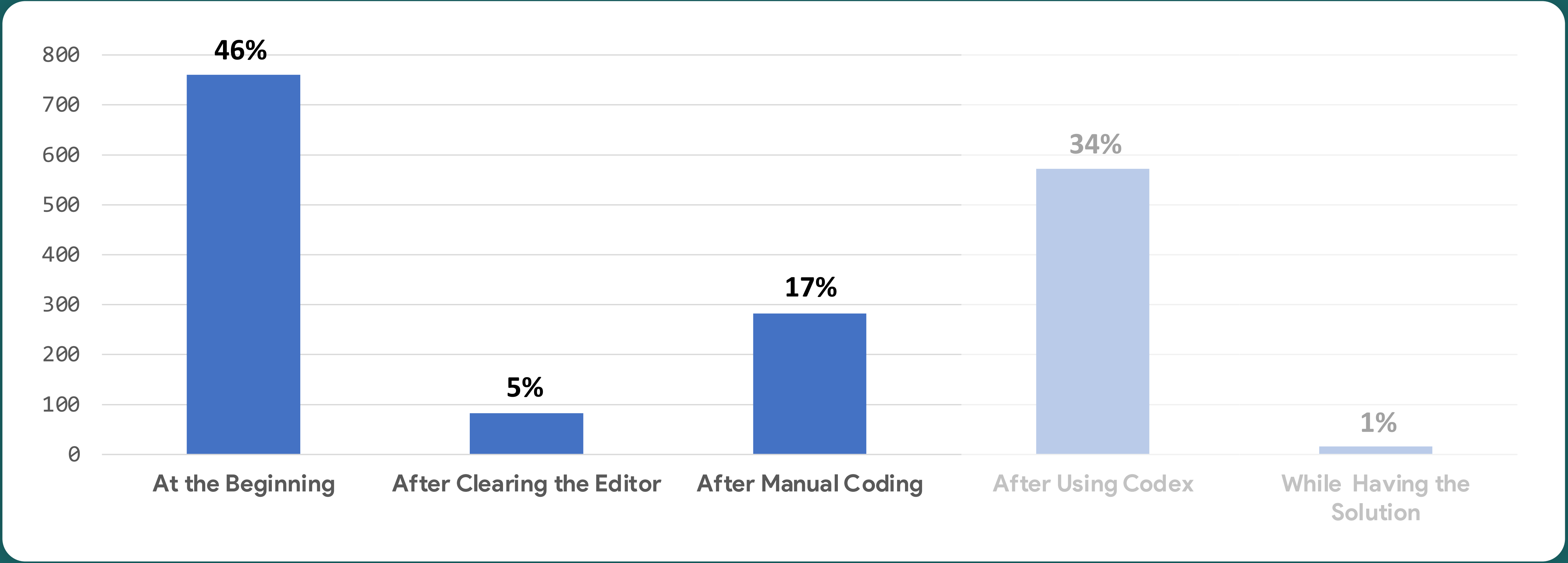
Five Primary Situations:



RQ1 A

When did Learners Use Codex?

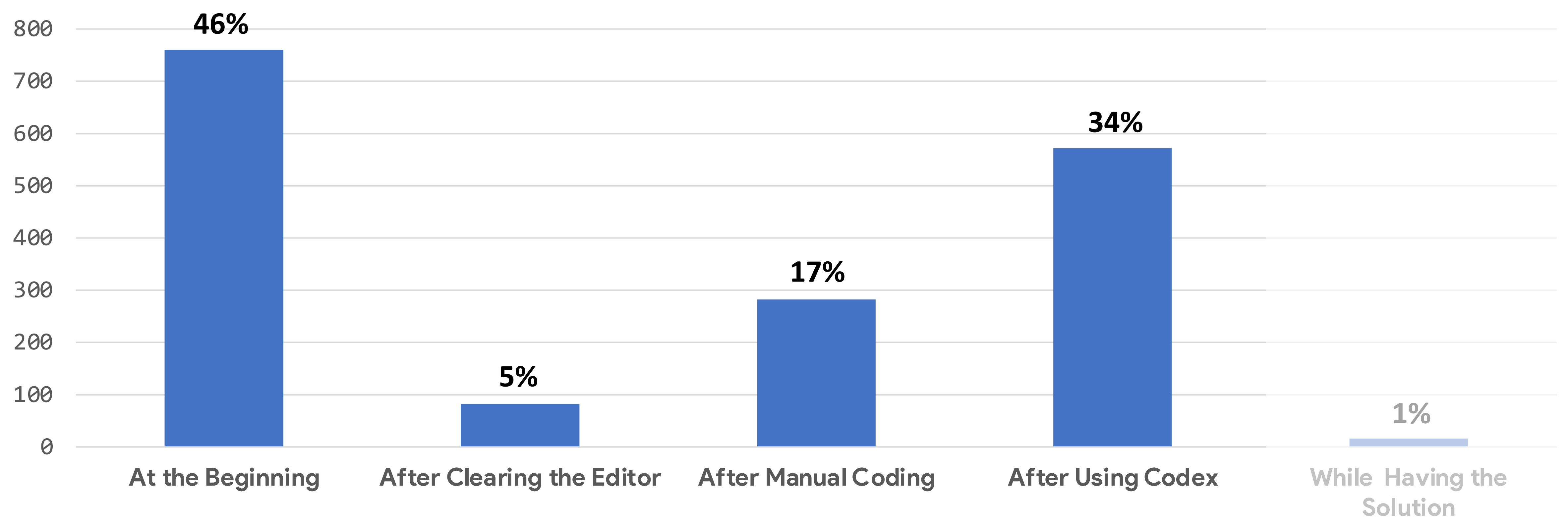
Five Primary Situations:



RQ1 A

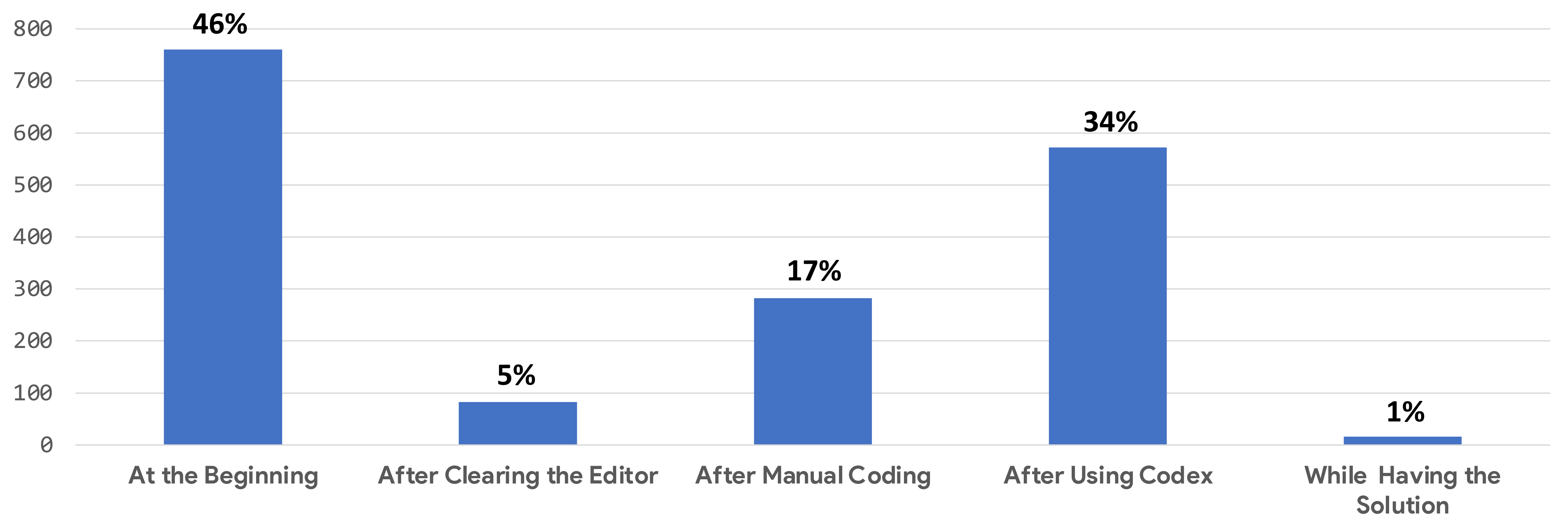
When did Learners Use Codex?

Five Primary Situations:



RQ1 A When did Learners Use Codex?

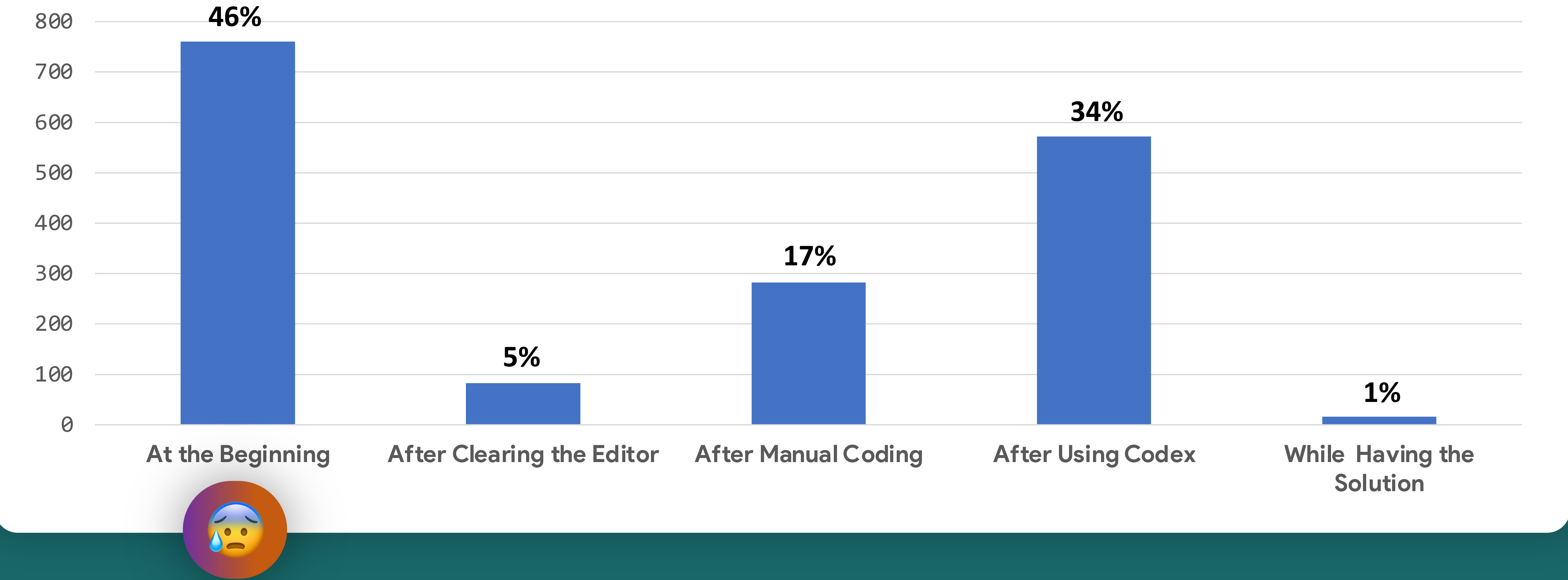
Five Primary Situations:



RQ1 A

When did Learners Use Codex?

Five Primary Situations:

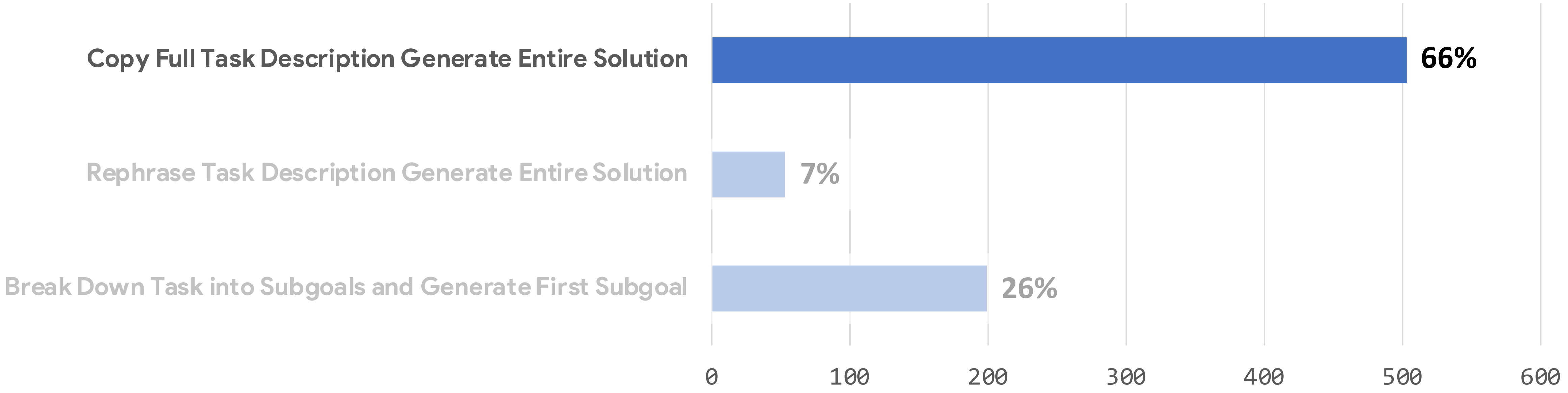


RQ1 A

When did Learners Use Codex?

Situation: Starting with Codex (n=760, 46%)

Common Behaviors **When Using Codex at The Beginning:**

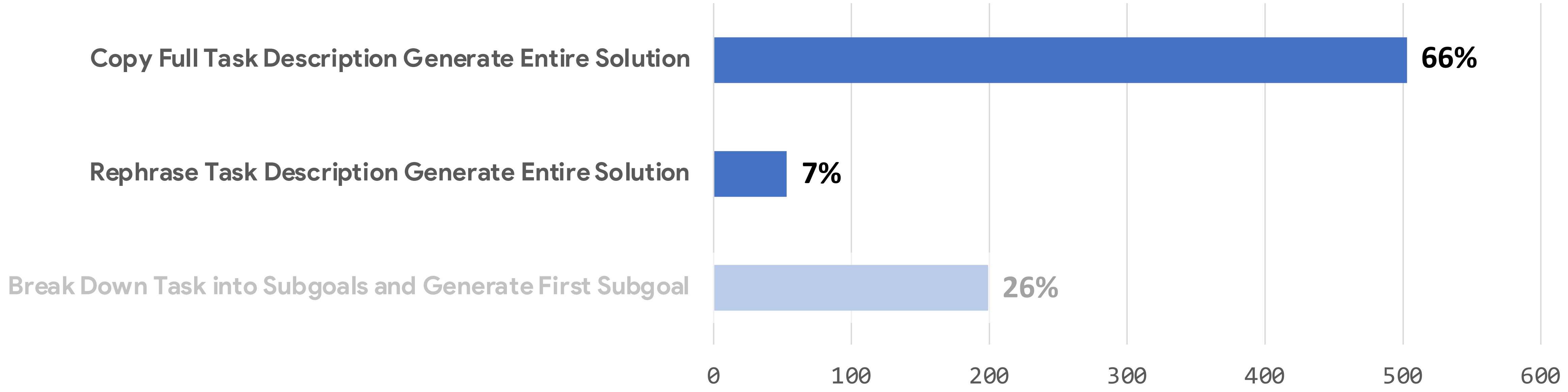


RQ1 A

When did Learners Use Codex?

Situation: Starting with Codex (n=760, 46%)

Common Behaviors **When Using Codex at The Beginning:**

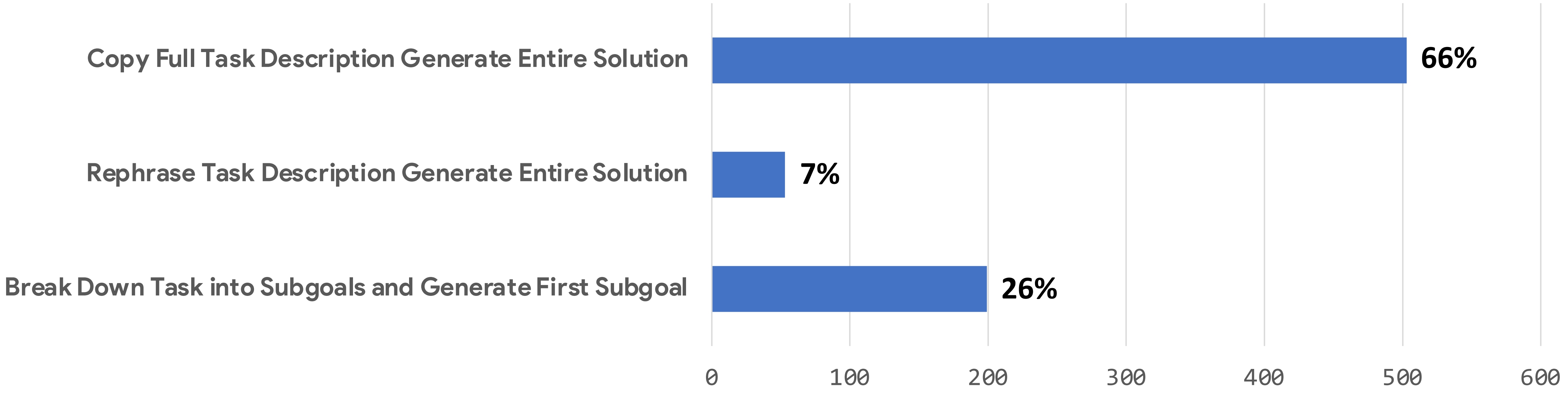


RQ1 A

When did Learners Use Codex?

Situation: Starting with Codex (n=760, 46%)

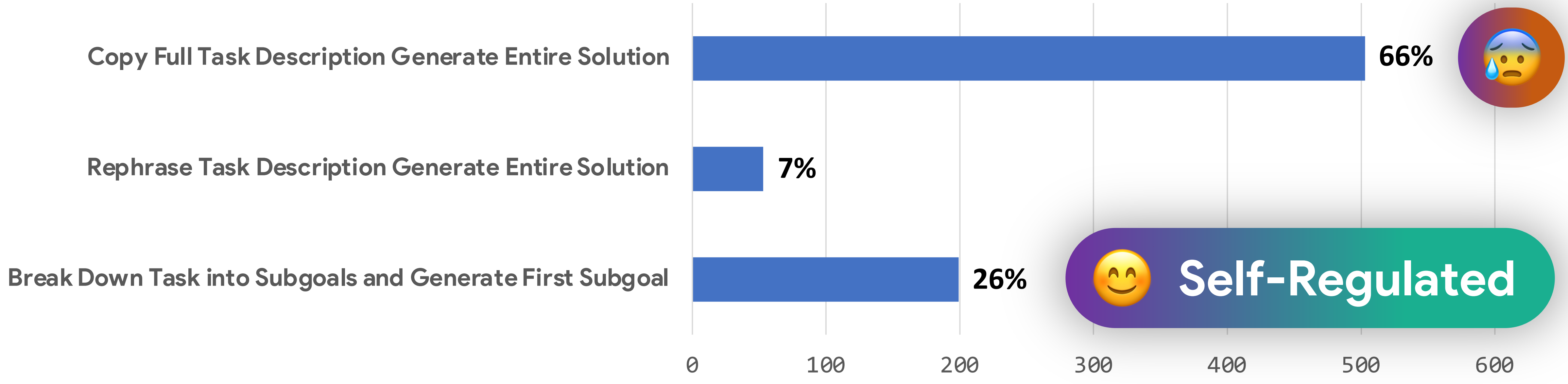
Common Behaviors **When Using Codex at The Beginning:**



RQ1 A When did Learners Use Codex?

Situation: Starting with Codex (n=760, 46%)

Common Behaviors **When Using Codex at The Beginning:**

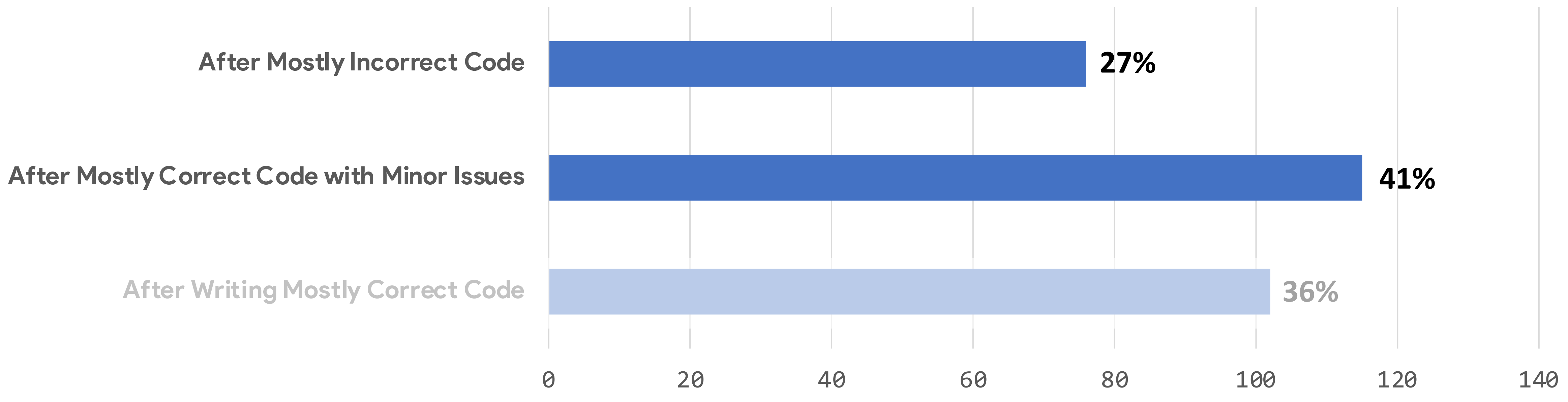


RQ1 A

When did Learners Use Codex?

Situation: After Manual Coding (n=282, 17%)

State of Code **When Using Codex After Manual Coding:**

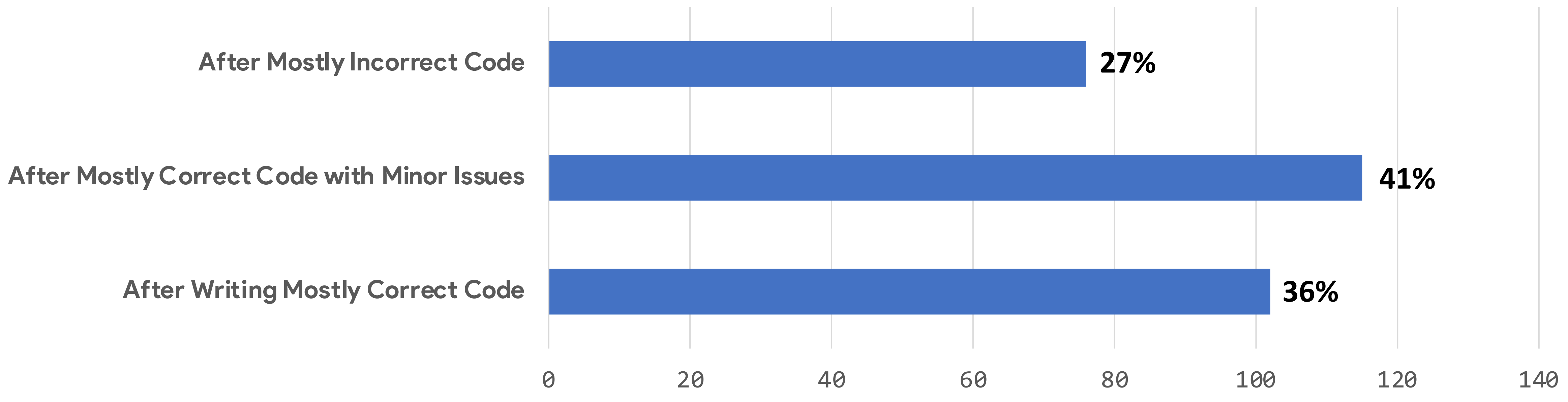


RQ1 A

When did Learners Use Codex?

Situation: After Manual Coding (n=282, 17%)

State of Code When Using Codex After Manual Coding:

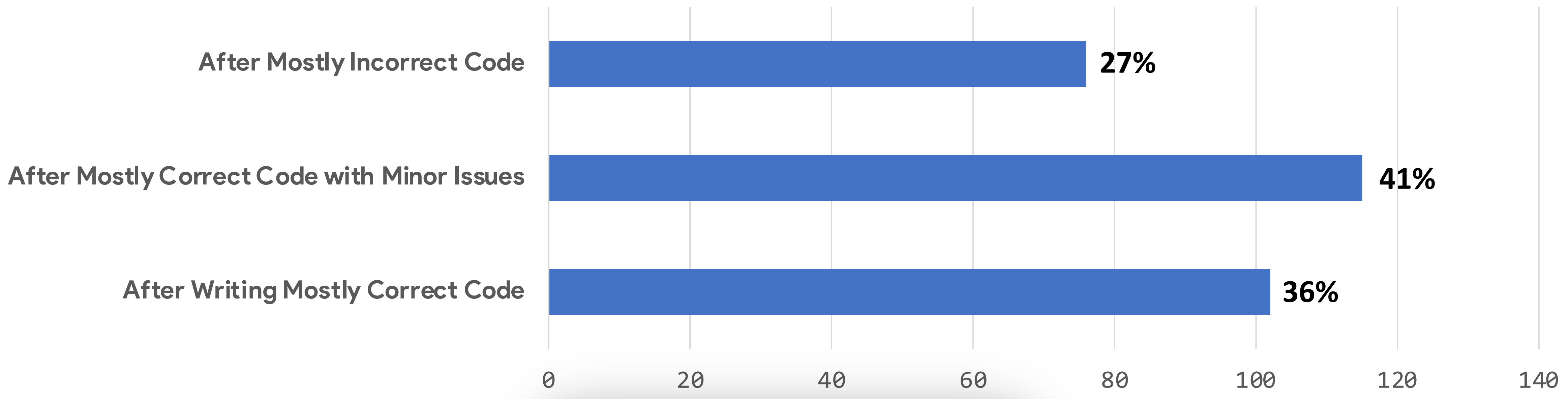


RQ1 A

When did Learners Use Codex?

Situation: After Manual Coding (n=282, 17%)

State of Code **When Using Codex After Manual Coding:**



😊 Self-Regulated

RQ1 A

When did Learners Use Codex?

Situation: After Using Codex (n=572, 34%)

- Decomposing Tasks into Multiple Subgoals: **Write Next Subgoal with Codex**

243 Codex Usages (15%)

1
```
import random
pivot = random.randint(1, 100)
```

2
```
# PROMPT: ask the user to enter a number
num = int(input("guess a number"))
```

84 Codex Usages (5%)

1
```
import random
num1 = random.randint(1, 6)
```

2
```
# PROMPT: generate another random number
num2 = random.randint(1, 6)
```

RQ1 A

When did Learners Use Codex?

Situation: After Using Codex (n=572, 34%)

- Decomposing Tasks into Multiple Subgoals: **Write Next Subgoal with Codex**

243 Codex Usages (15%)

1
```
import random
pivot = random.randint(1, 100)
```

2
```
# PROMPT: ask the user to enter a number
num = int(input("guess a number"))
```

84 Codex Usages (5%)

1
```
import random
num1 = random.randint(1, 6)
```

2
```
# PROMPT: generate another random number
num2 = random.randint(1, 6)
```

😰 Over-Reliance

RQ1 A

When did Learners Use Codex?

Situation: Already Having the Solution (n=16, 1%)

Have Solution

↓

Use Codex to Generate Solution

↓

Check and Edit Own Solution

😊 Self-Evaluation

Results

What did Learners Ask from Codex?

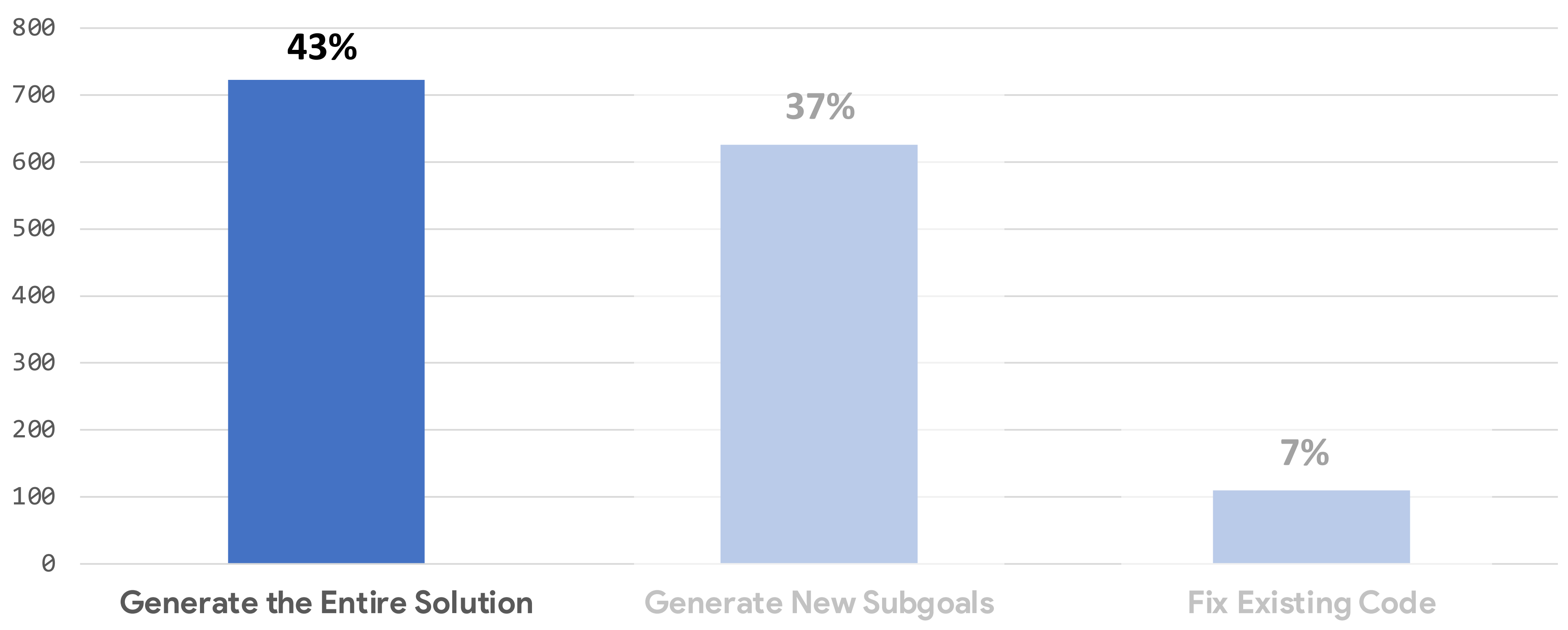
Focus of Thematic Analysis:

- What parts of the task?
- Requesting Syntax or Logic?

RQ1 B

What did Learners Ask from Codex?

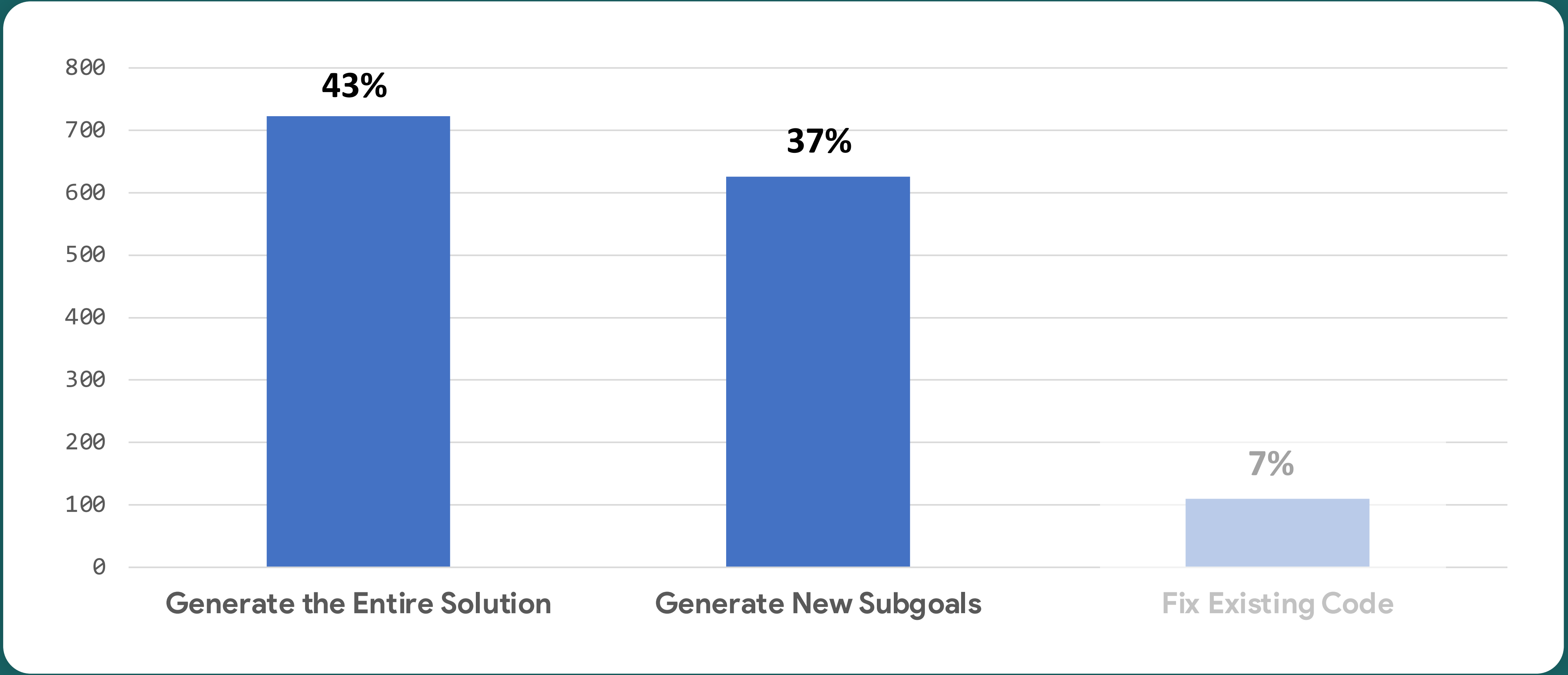
What are they asking for using Codex?



RQ1 B

What did Learners Ask from Codex?

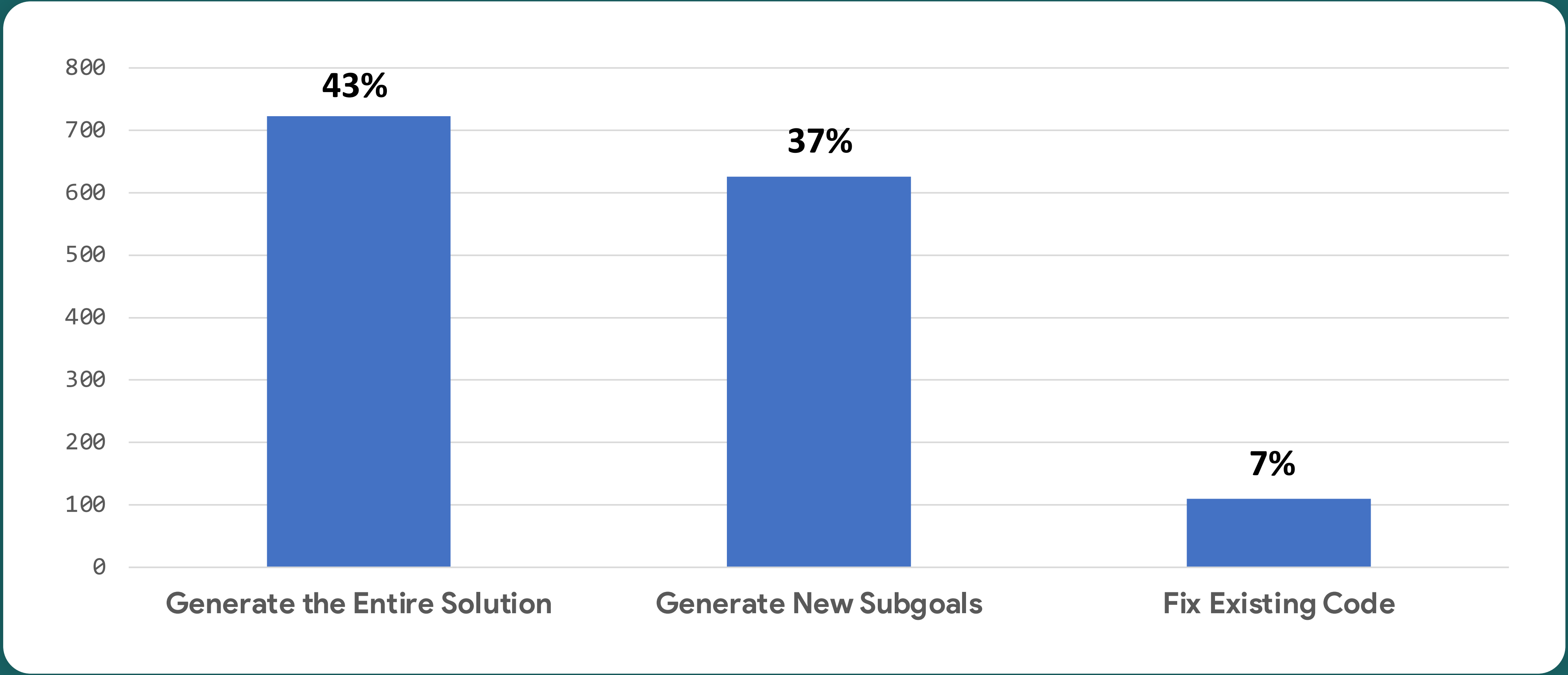
What are they asking for using Codex?



RQ1 B

What did Learners Ask from Codex?

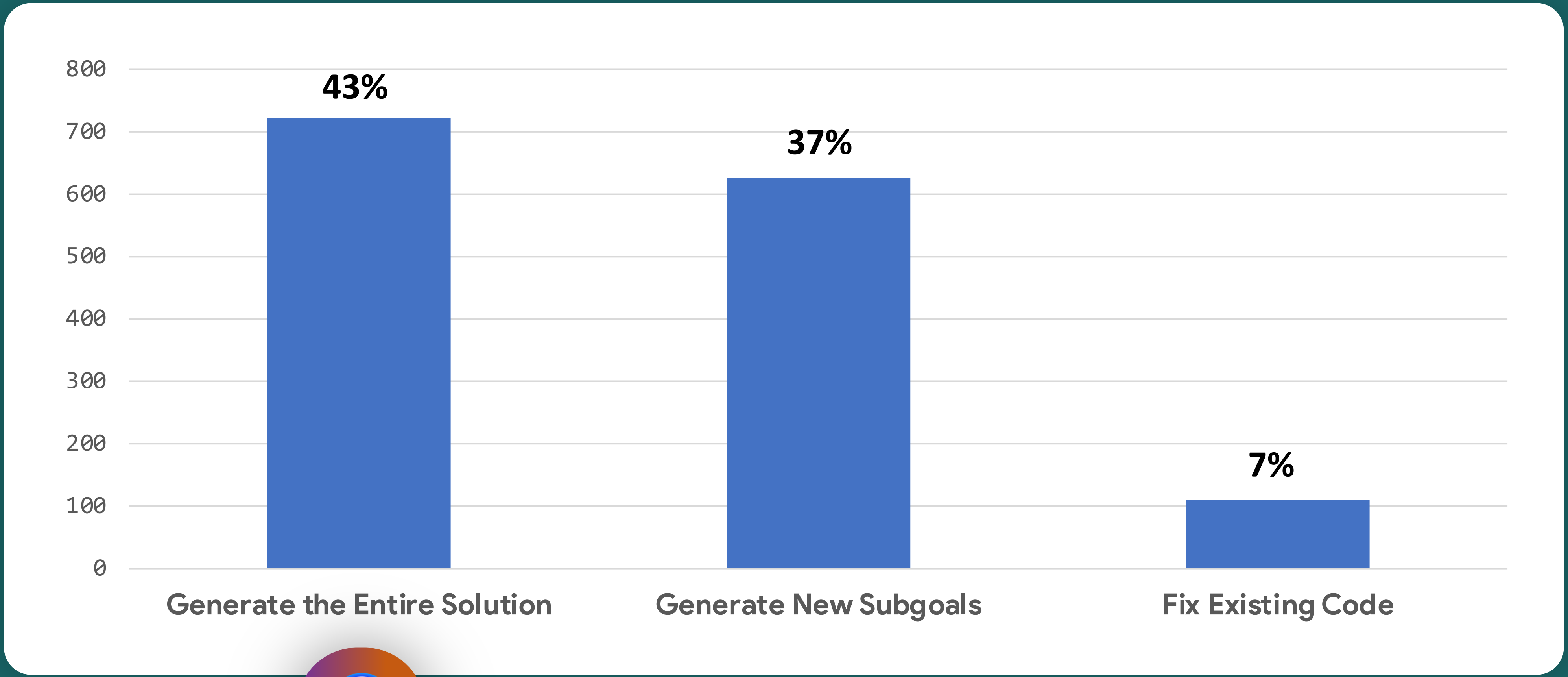
What are they asking for using Codex?



RQ1 B

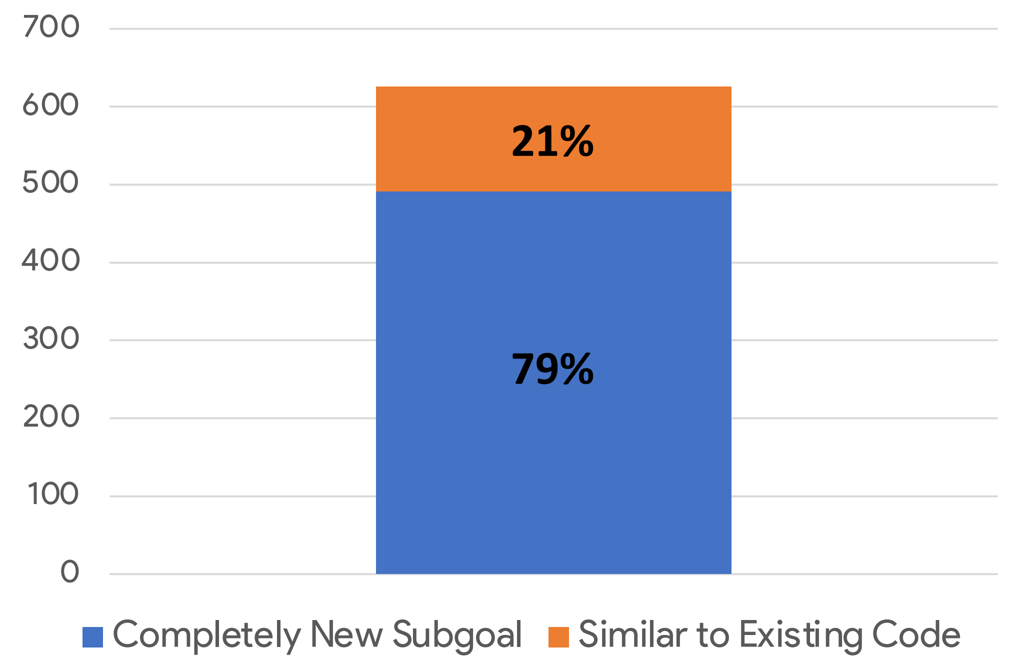
What did Learners Ask from Codex?

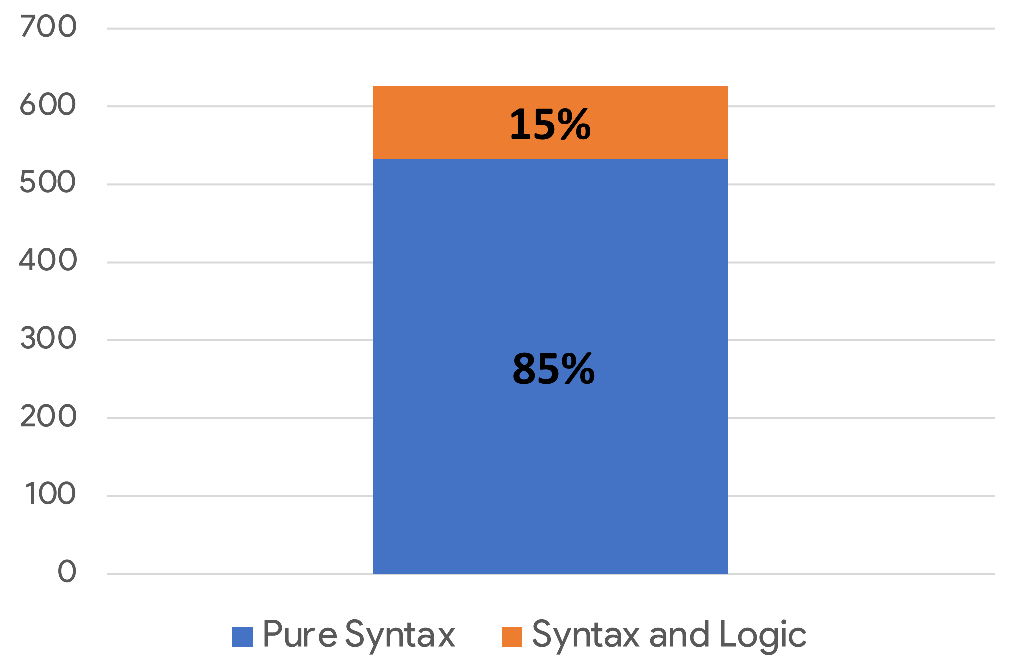
What are they asking for using Codex?



RQ1 B What did Learners Ask from Codex?

When Decomposing Task into Subgoals

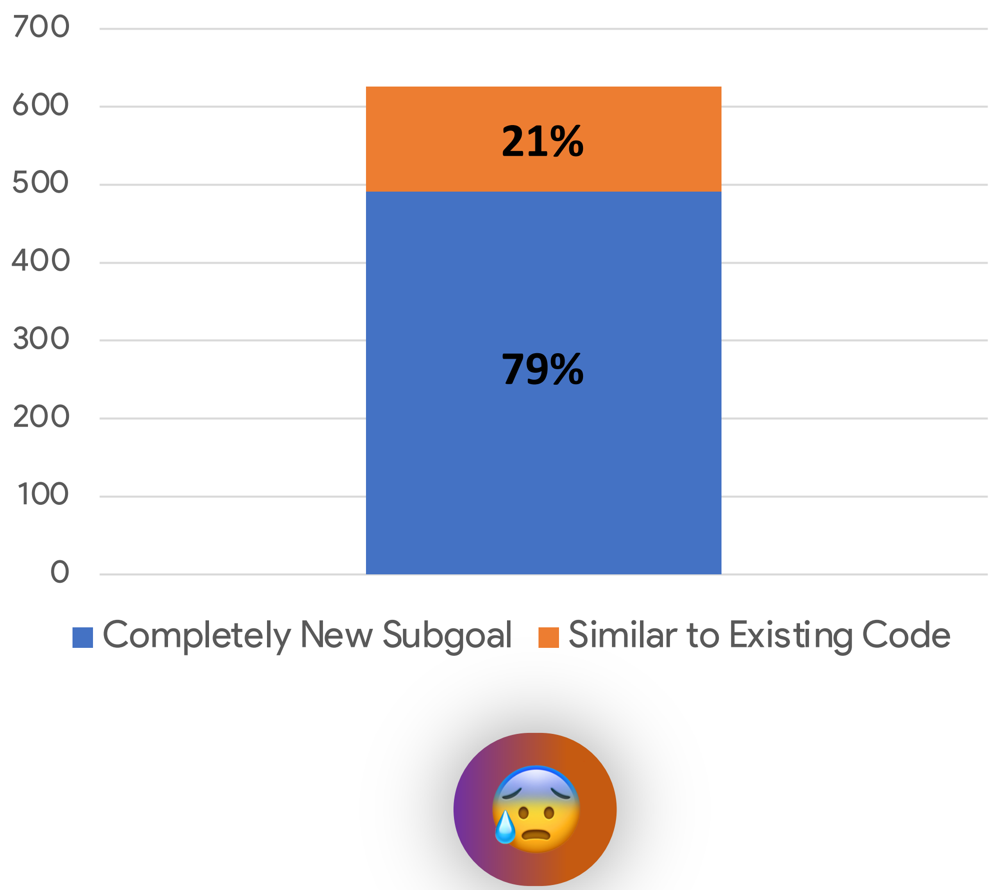


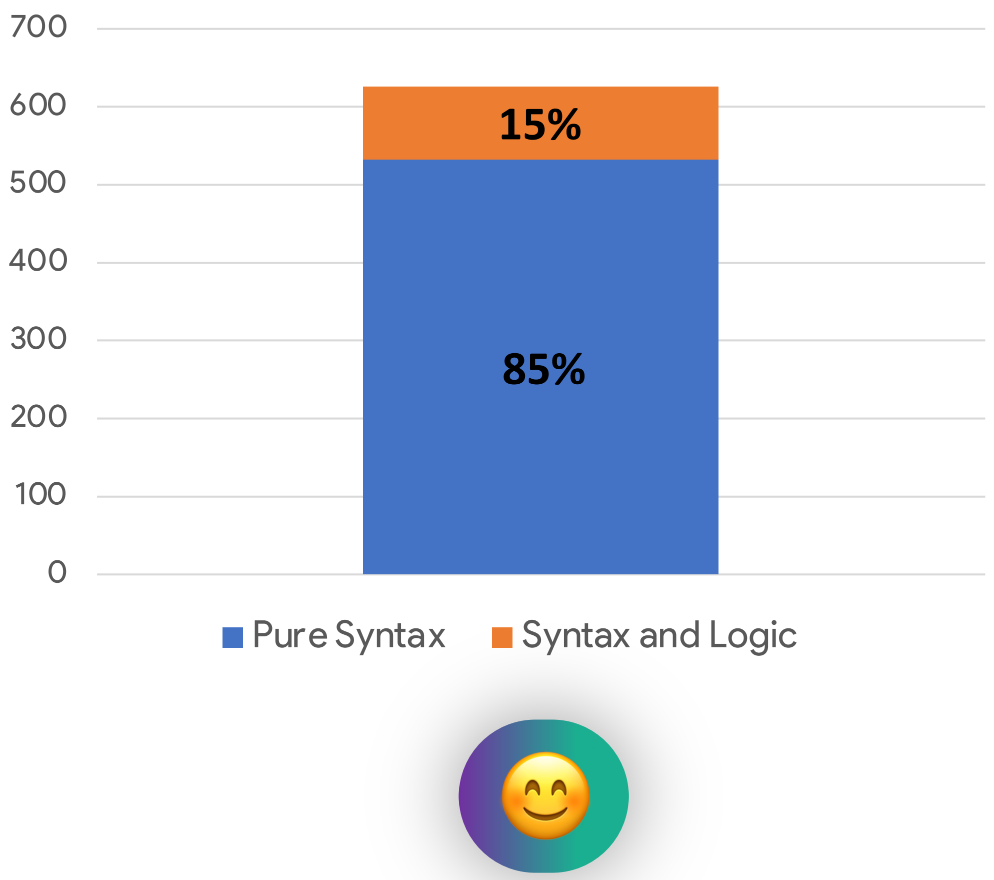


RQ1 B

What did Learners Ask from Codex?

When Decomposing Task into Subgoals





Results

Novice Learners' Prompt Properties

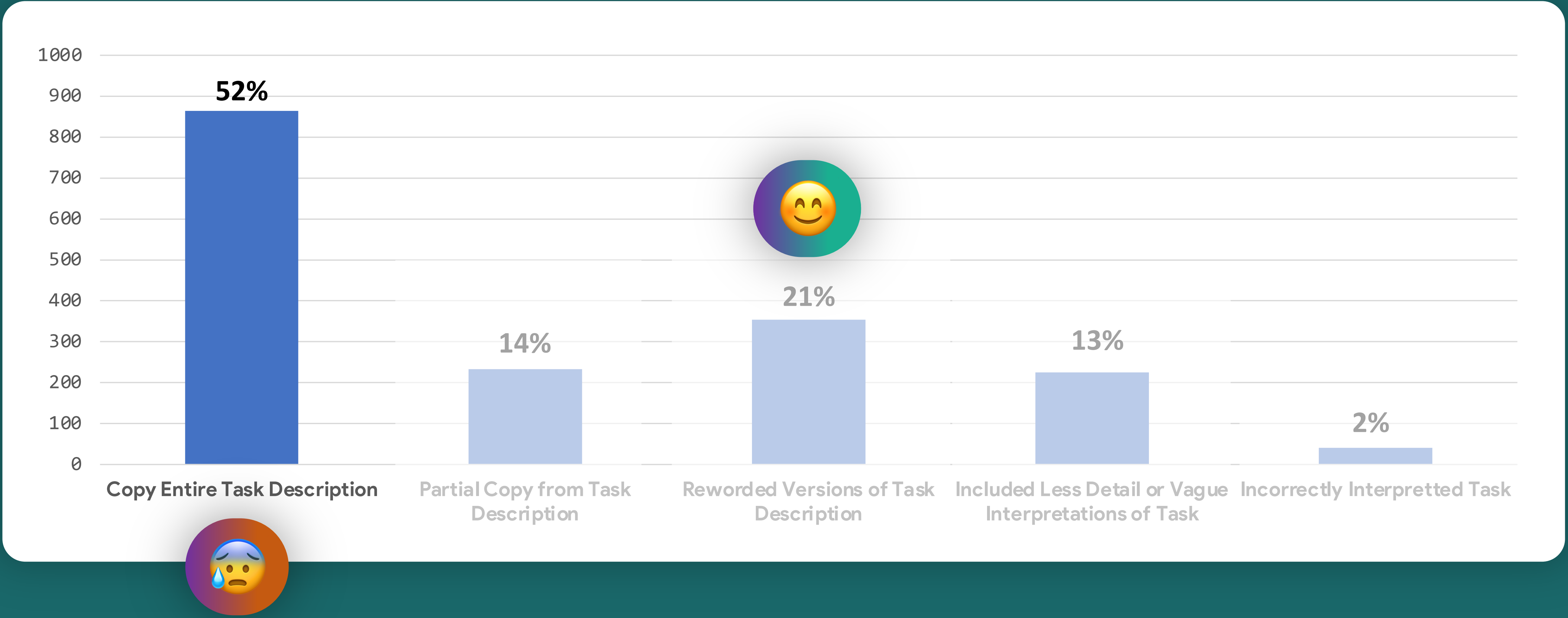
Focus of Thematic Analysis:

- Prompt Content
- Vagueness
- Relationship to Task Description

RQ1 C

Novice Learners' Prompt Properties

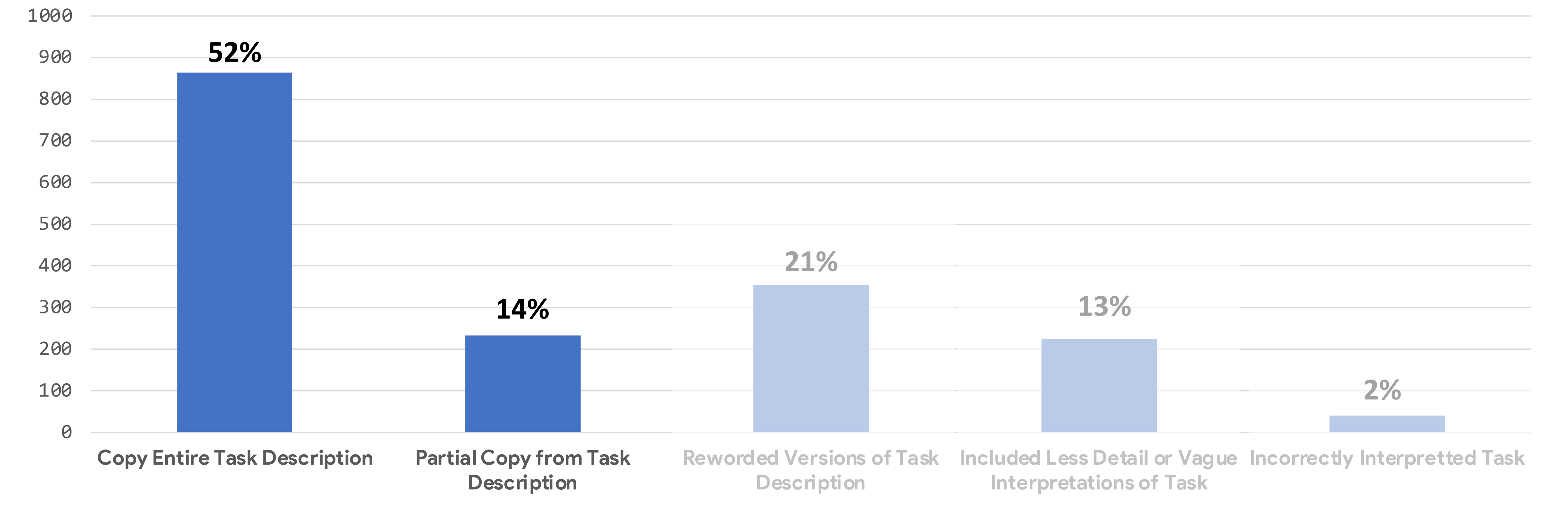
Five Primary Properties:



RQ1 C

Novice Learners' Prompt Properties

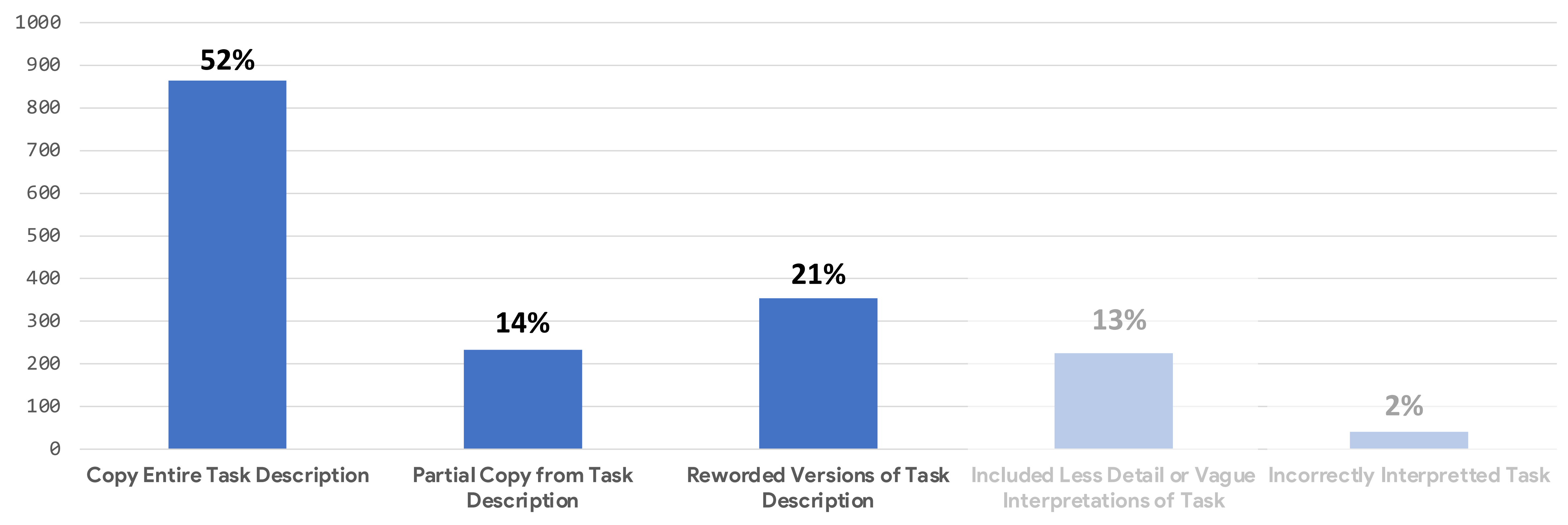
Five Primary Properties:



RQ1 C

Novice Learners’ Prompt Properties

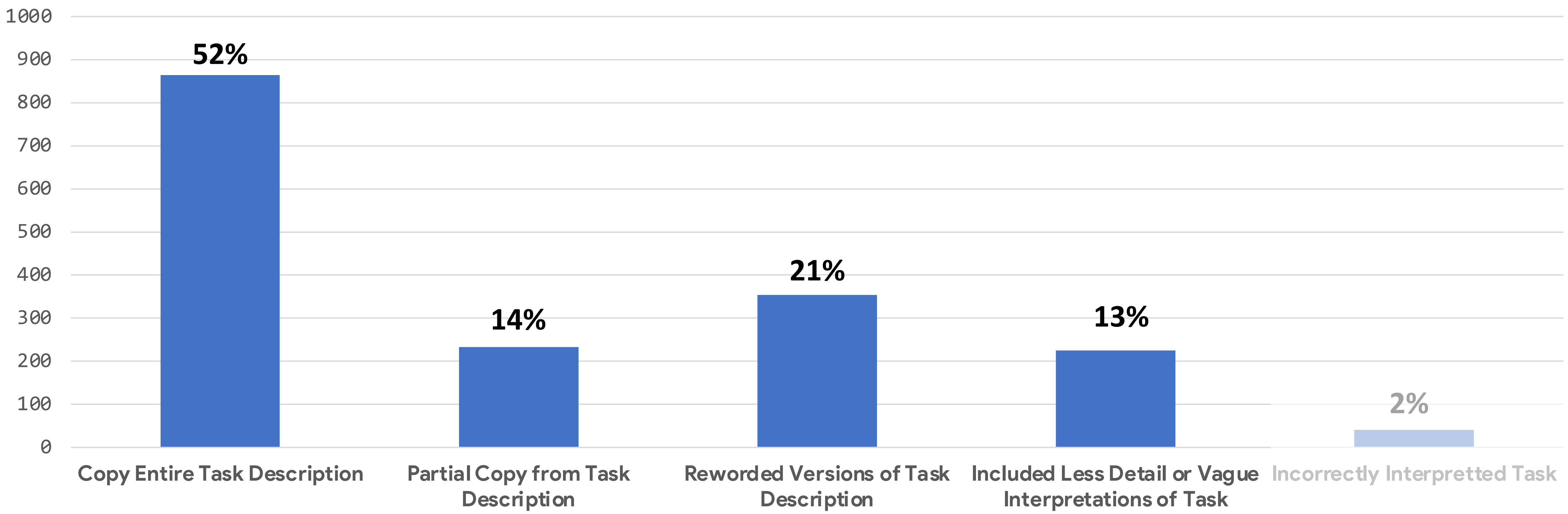
Five Primary Properties:



RQ1 C

Novice Learners' Prompt Properties

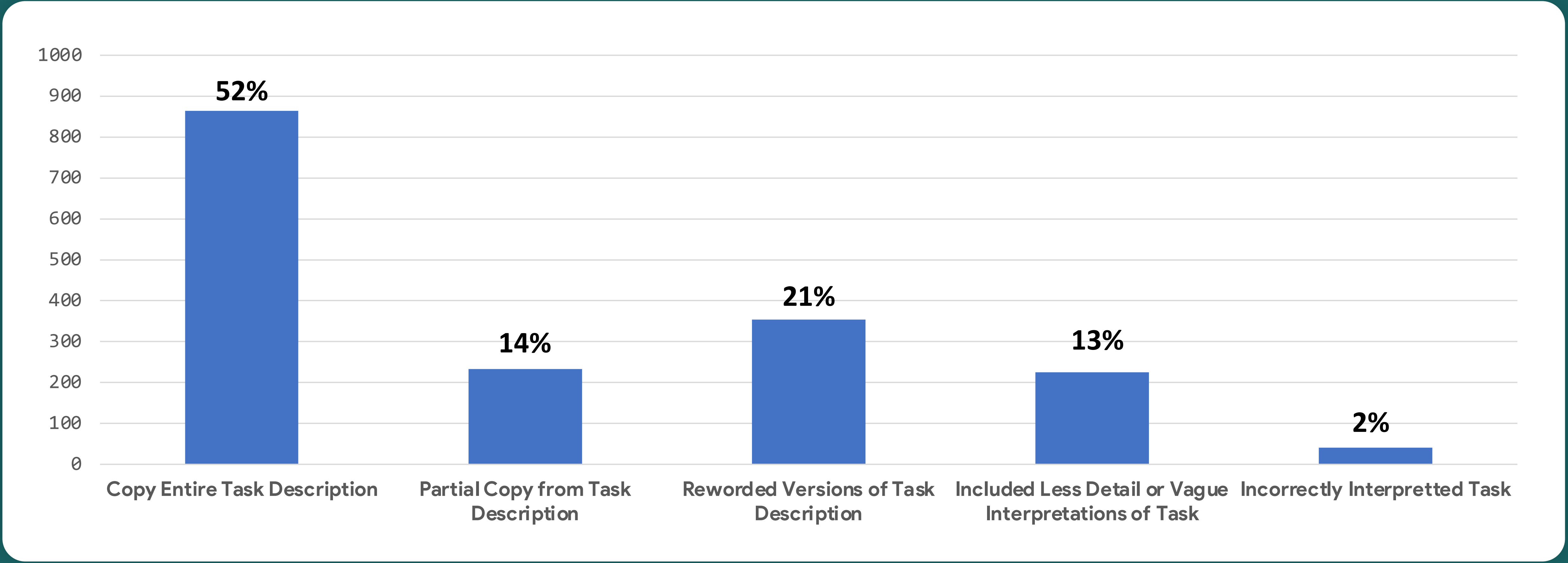
Five Primary Properties:



RQ1 C

Novice Learners' Prompt Properties

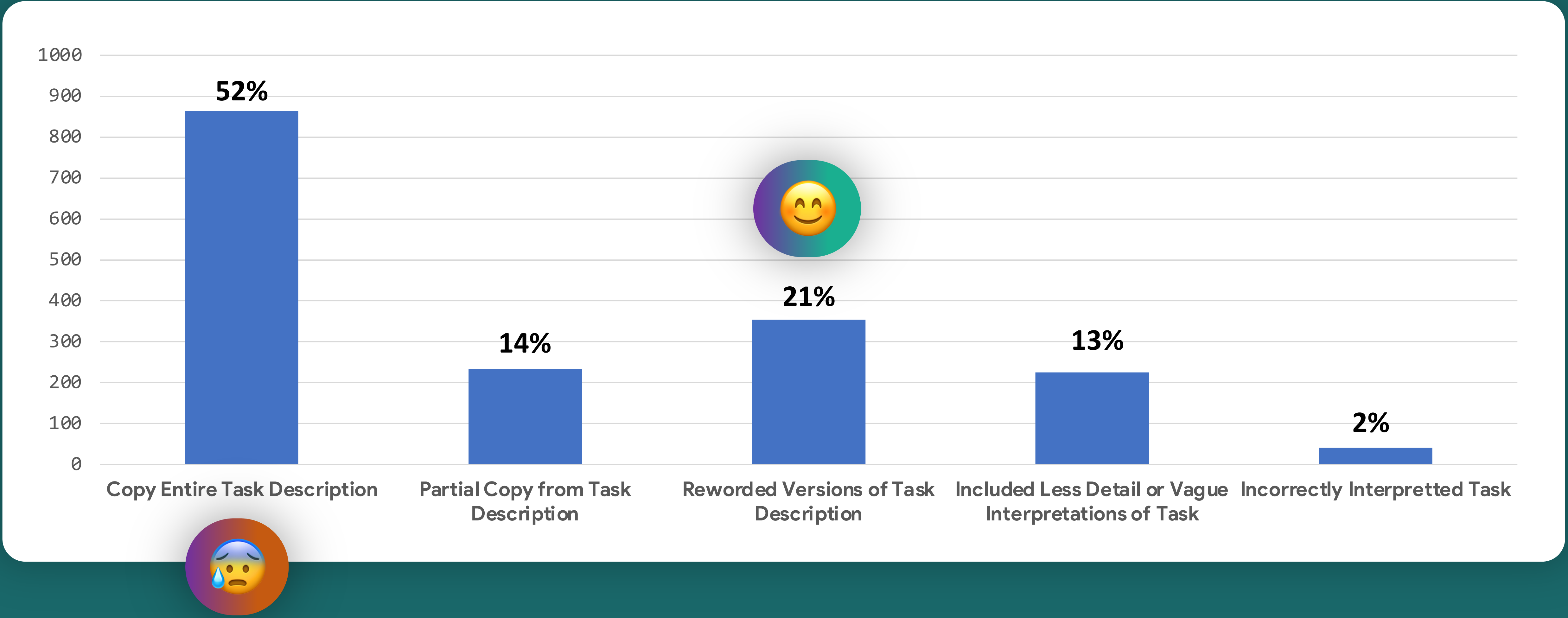
Five Primary Properties:



RQ1 C

Novice Learners' Prompt Properties

Five Primary Properties:



RQ1 C

Novice Learners' Prompt Properties

Prompts Similar to Pseudo-Code (n=89, 5%)

Prompt 1: "for n in numbers, if n > l, set l to n"

↓

Prompt 2: "print Largest number: l"

😊 Self-Regulated

Prompt: "find the largest number"

Results

Utilizing AI-Generated Code

Focus of Thematic Analysis:

- Placement of AI-Generated Code
- Modifying Existing or Generated Code
- Testing and Verifying Code

RQ1 E

Utilizing AI-Generated Code

Verifying: **Tinkering with AI-Generated Code**

```
# PROMPT: print 1st message in list
print(numbers[0])
```

↓

```
print(numbers[1])
```

↓

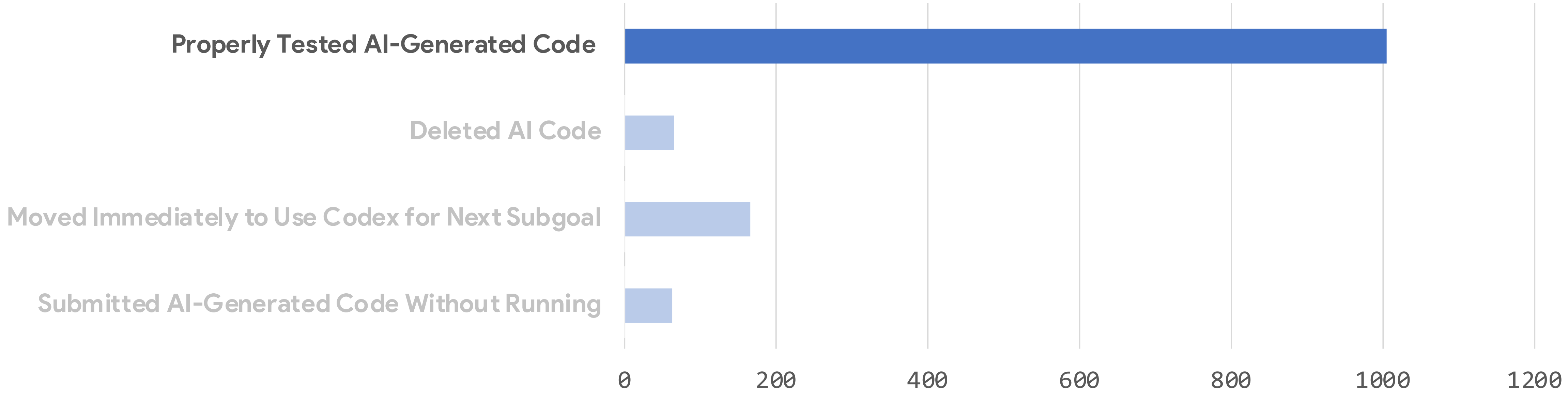
```
print(numbers[0])
```

RQ1 E

Utilizing AI-Generated Code

Verifying: **Running and Testing AI-Generated Code**

Common Behaviors **When Using Codex at The Beginning:**

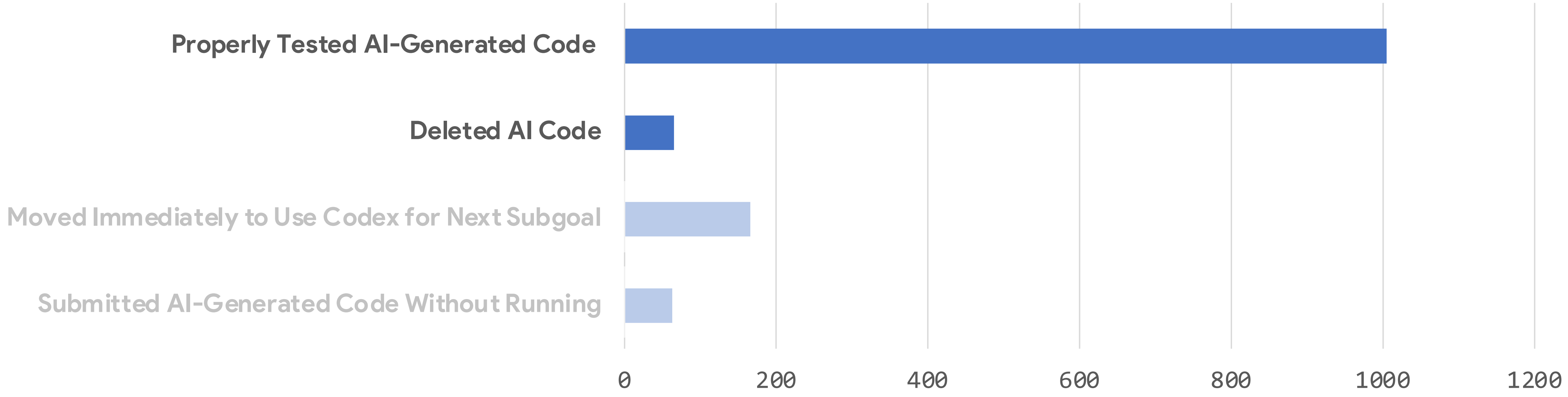


RQ1 E

Utilizing AI-Generated Code

Verifying: **Running and Testing AI-Generated Code**

Common Behaviors **When Using Codex at The Beginning:**

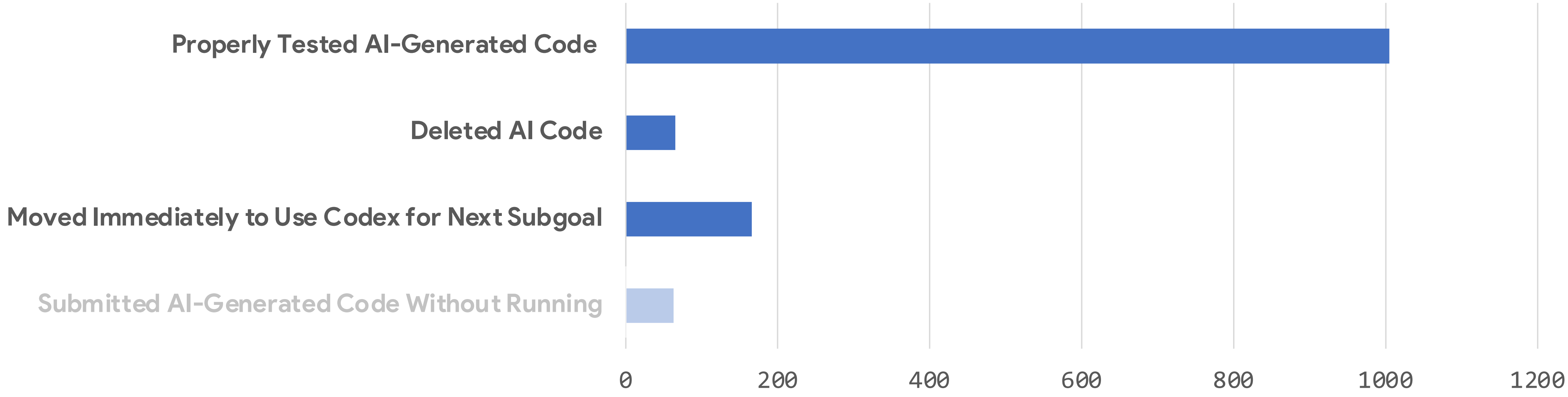


RQ1 E

Utilizing AI-Generated Code

Verifying: **Running and Testing AI-Generated Code**

Common Behaviors **When Using Codex at The Beginning:**

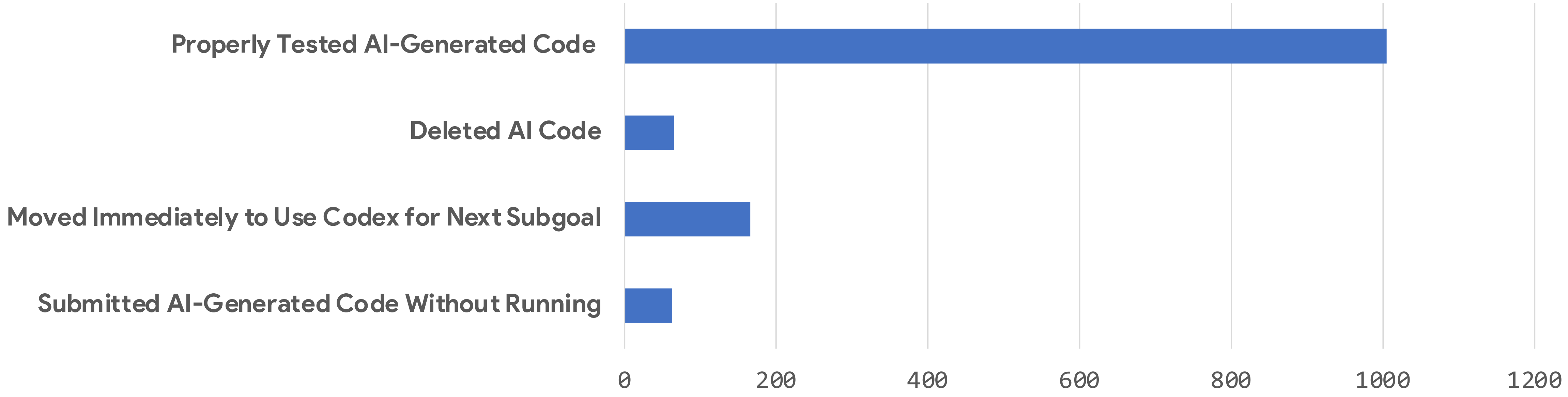


RQ1 E

Utilizing AI-Generated Code

Verifying: **Running and Testing AI-Generated Code**

Common Behaviors **When Using Codex at The Beginning:**

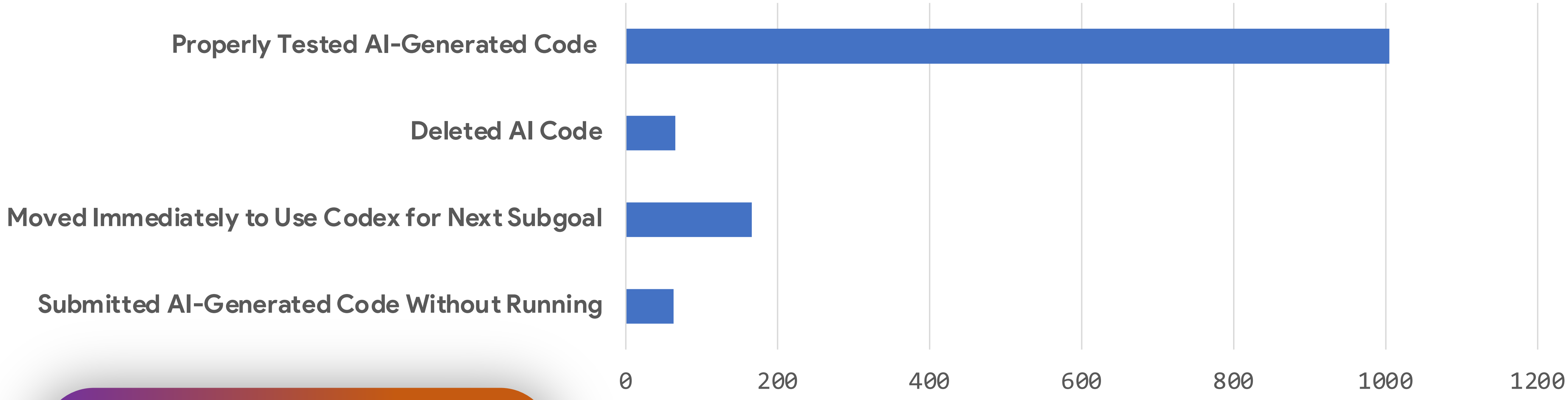


RQ1 E

Utilizing AI-Generated Code

Verifying: **Running and Testing AI-Generated Code**

Common Behaviors **When Using Codex at The Beginning:**



😰 Over-Reliance

RQ1 E

Utilizing AI-Generated Code

Verifying: **Manually Adding Code to Verify**

1. Prompt Codex

```
> generate two
random numbers
between 1 and 6
and check both
if they are
greater than 3
```

OpenAI Codex

2. Generated Code + Placed

```python
import random
roll1 = random.randint(1, 6)
roll2 = random.randint(1, 6)
if roll1 > 3 and roll2 > 3:
        print("Both greater than 3")
```

3. Added Verification Code

```python
import random
roll1 = random.randint(1, 6)
roll2 = random.randint(1, 6)
print(roll1, ",",roll2)
if roll1 > 3 and roll2 > 3:
        print("Both greater than 3")
```

Student added a new line
`print(roll1, ",",roll2)`
to verify the AI-generated code.

RQ1 E

Utilizing AI-Generated Code

Verifying: **Manually Adding Code to Verify**

1. Prompt Codex

```
> generate two
random numbers
between 1 and 6
and check both
if they are
greater than 3
```

OpenAI Codex

2. Generated Code + Placed

```python
import random
roll1 = random.randint(1, 6)
roll2 = random.randint(1, 6)
if roll1 > 3 and roll2 > 3:
        print("Both greater than 3")
```

3. Added Verification Code

```python
import random
roll1 = random.randint(1, 6)
roll2 = random.randint(1, 6)
print(roll1, ",",roll2)
if roll1 > 3 and roll2 > 3:
        print("Both greater than 3")
```

Student added a new line
`print(roll1, ",",roll2)`
to verify the AI-generated code.

Self-Regulation

Results

AI Code Generator Coding Approaches

Manual (without Codex)

The final submitted code was 100% manually written.

29% tasks

Manual (without Codex)

The final submitted code was 100% manually written.

29% tasks

AI Step-by-Step

Decomposed task into multiple, consecutive Codex usages, with no manual coding

6% tasks

Manual (without Codex)

The final submitted code was 100% manually written.

29% tasks

AI Step-by-Step

Decomposed task into multiple, consecutive Codex usages, with no manual coding

6% tasks

Hybrid

A few subgoals were AI-generated, while other subgoals were written manually

19% tasks

Manual (without Codex)

The final submitted code was 100% manually written.

29% tasks

AI Step-by-Step

Decomposed task into multiple, consecutive Codex usages, with no manual coding

6% tasks

Hybrid

A few subgoals were AI-generated, while other subgoals were written manually

19% tasks

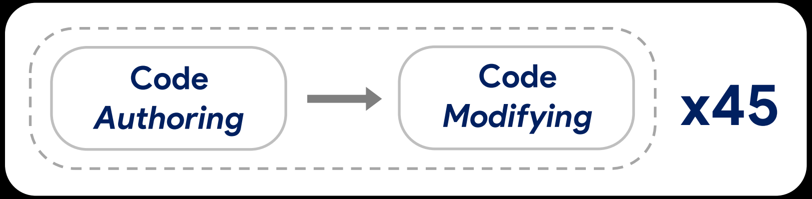
AI Single Prompt

Use a single prompt (either by copying the task, or rewording) to solve the entire task

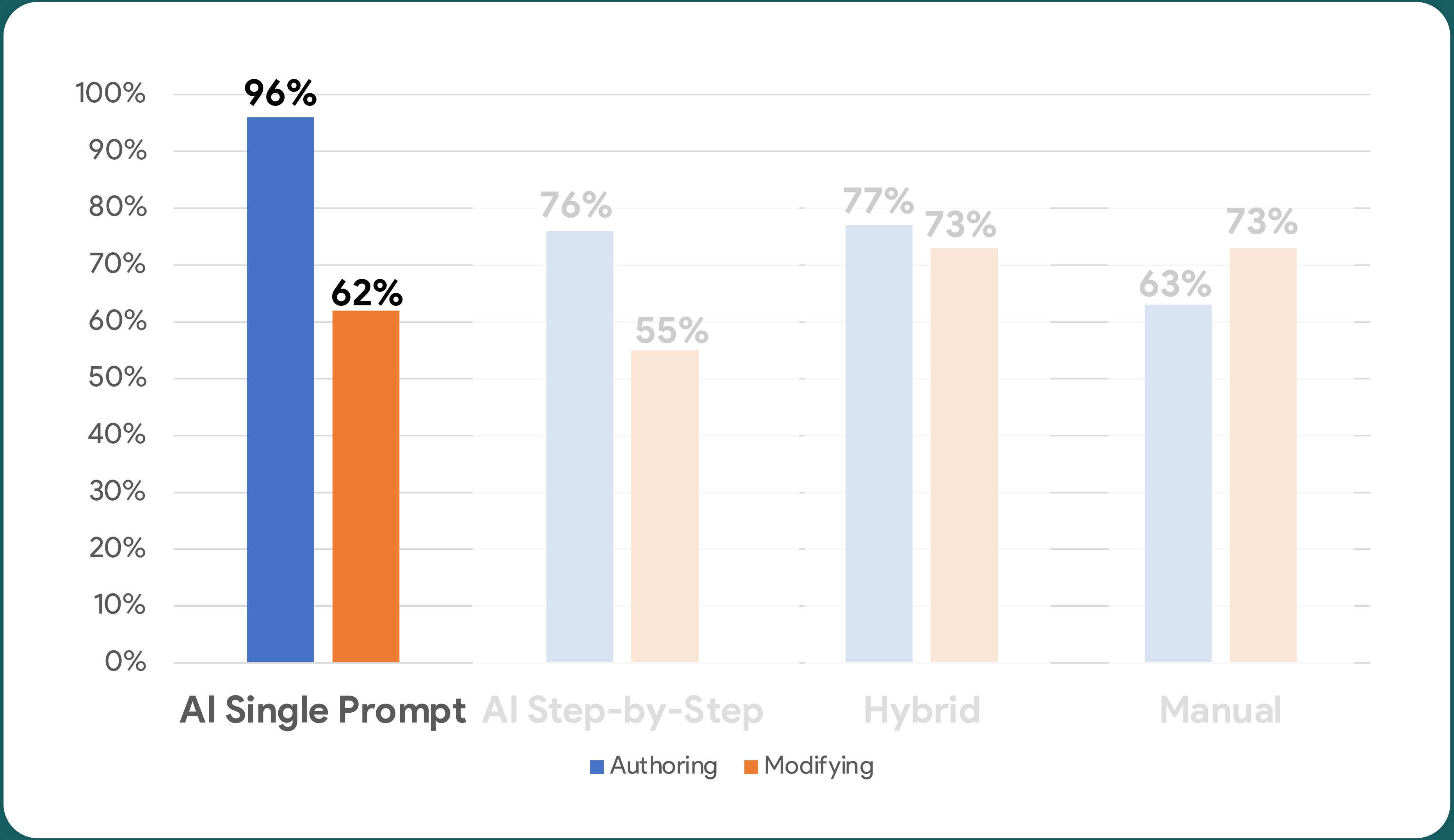
46% tasks

What is the Relationship between Authoring and Modifying Tasks for Each Coding Approach?

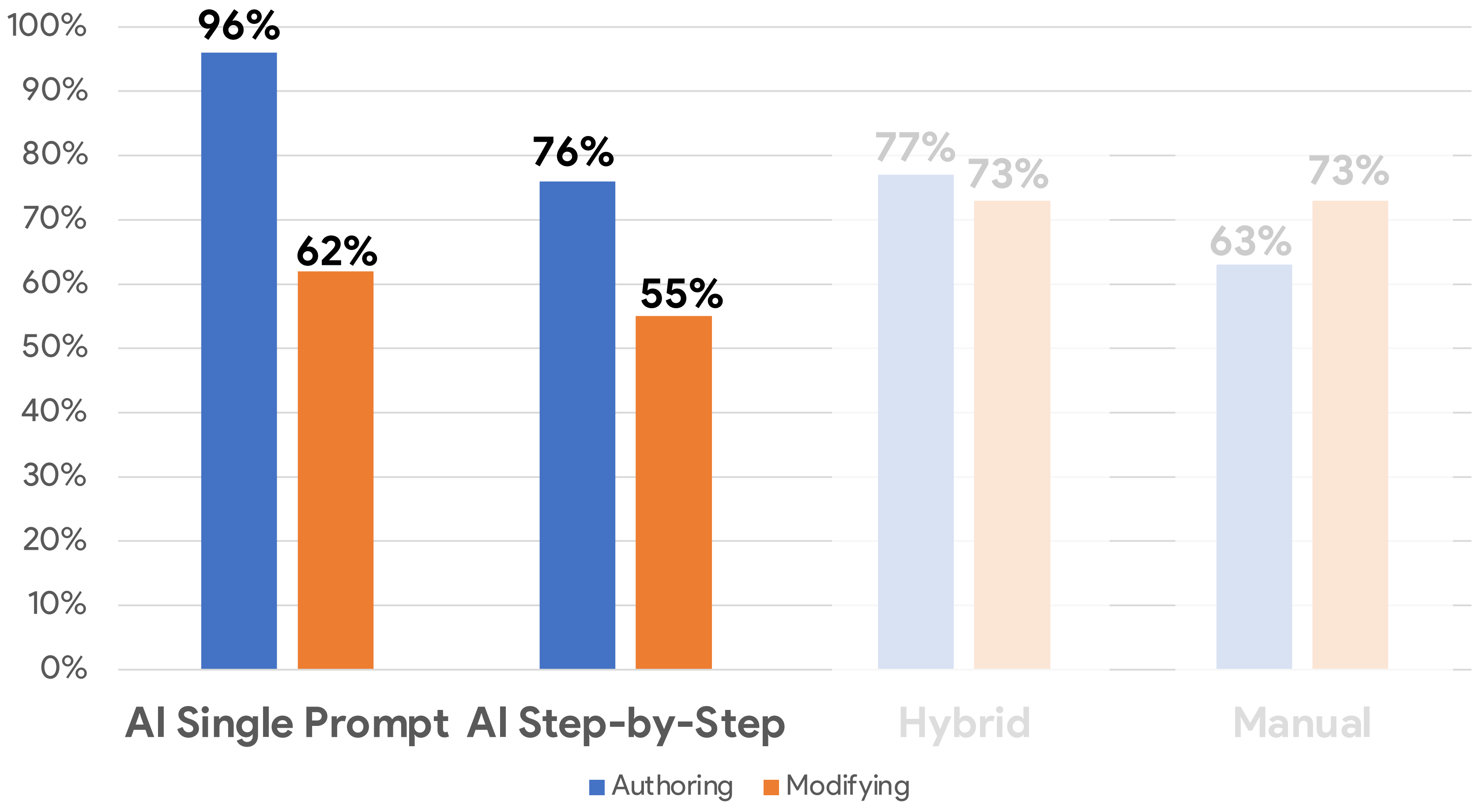
Self-Paced Python Training
7 Sessions



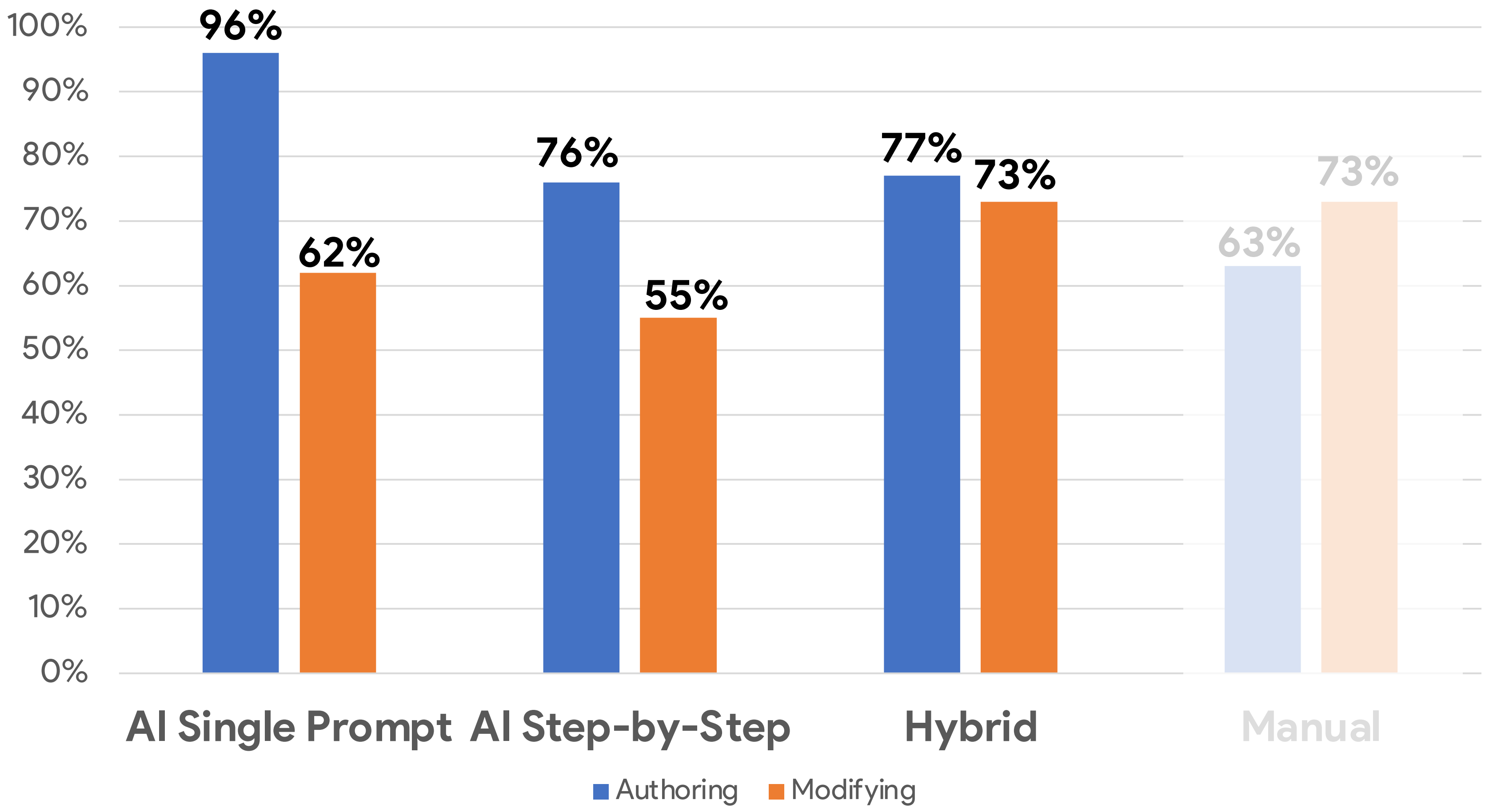
Relationship between Authoring and Modifying Tasks



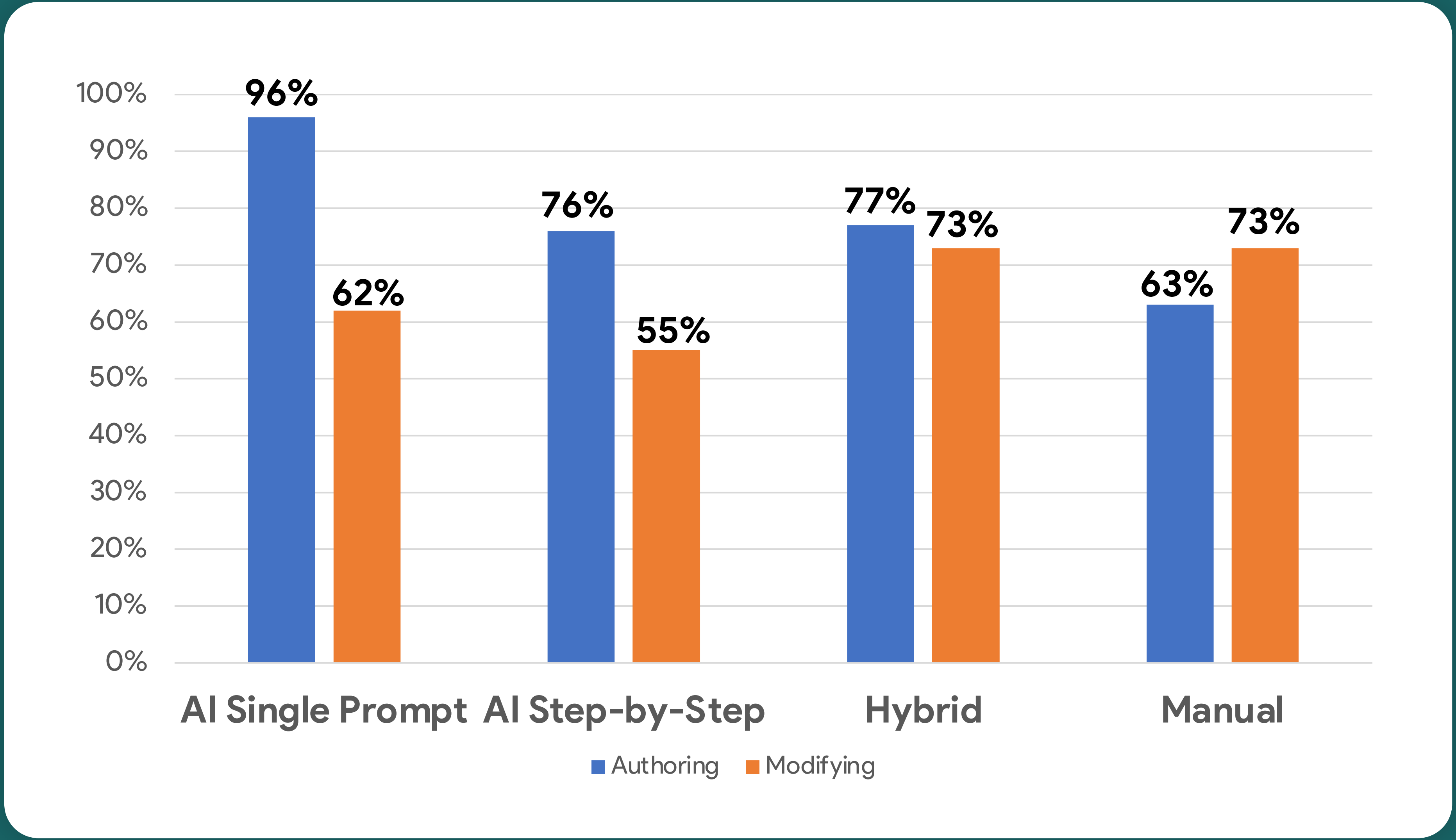
Relationship between Authoring and Modifying Tasks



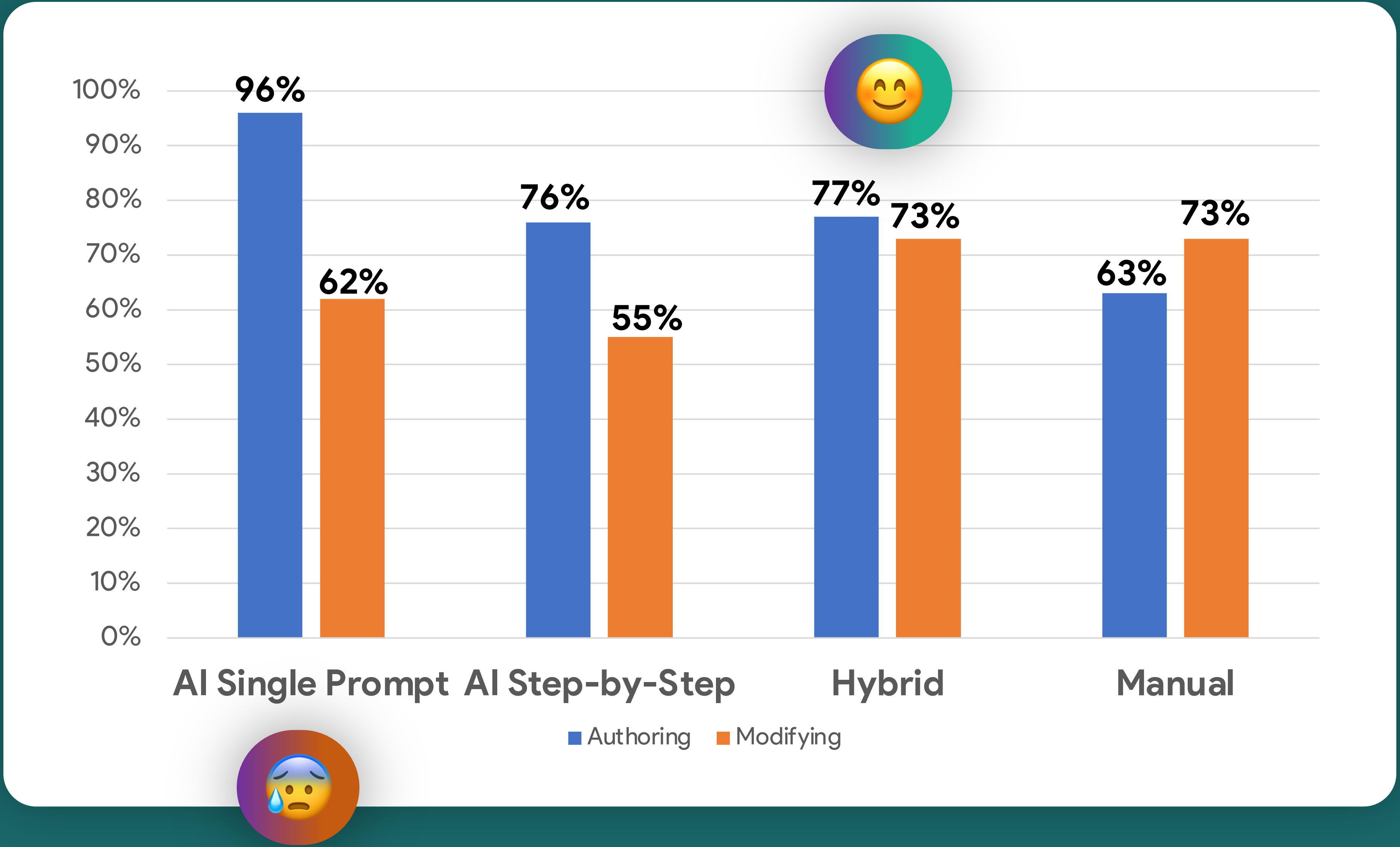
Relationship between Authoring and Modifying Tasks



Relationship between Authoring and Modifying Tasks



Relationship between Authoring and Modifying Tasks



What is the Relationship between utilizing each of the Coding Approaches and Post-Test Evaluation Tests?

1. Immediate Post-Test

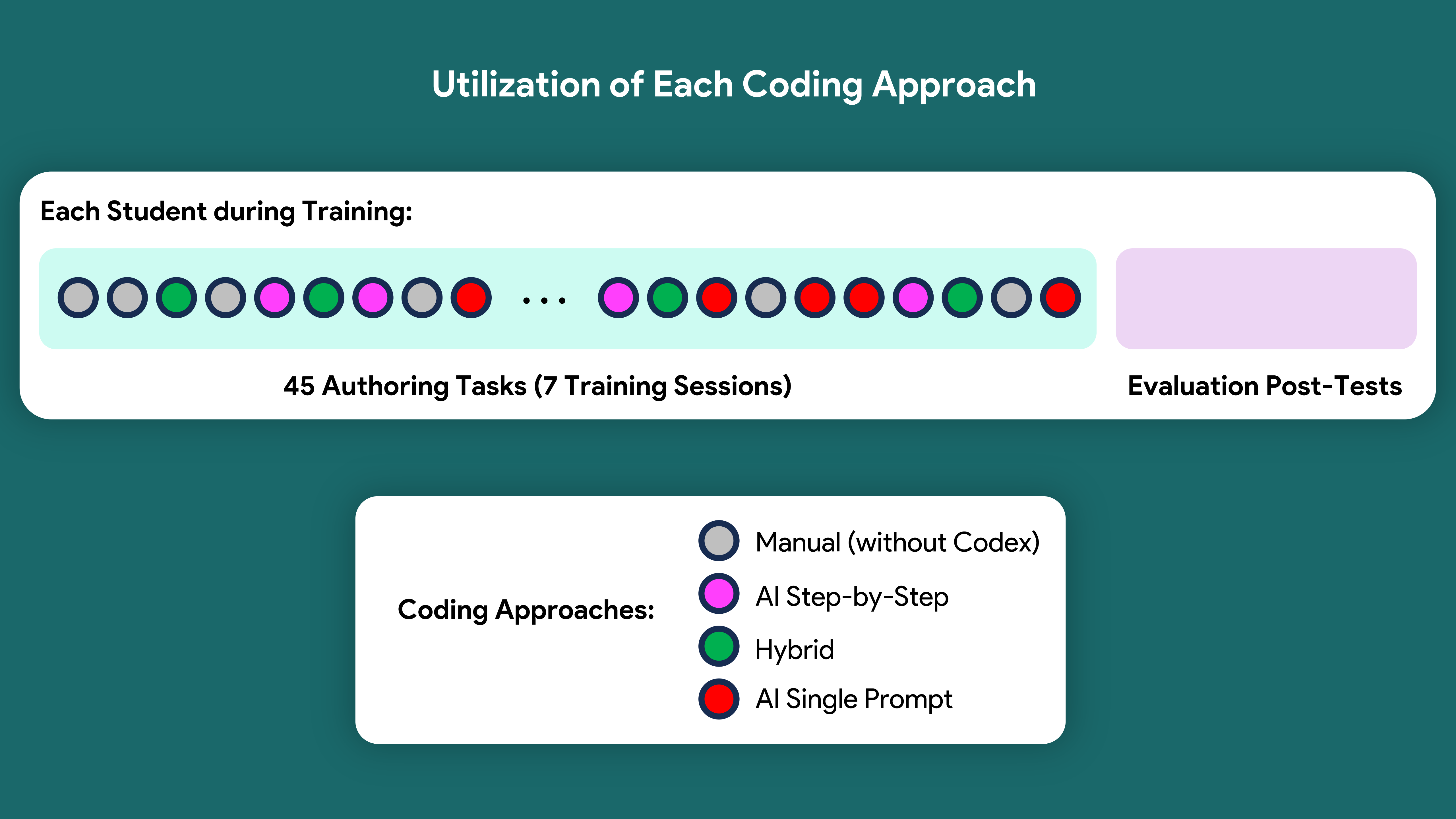
- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

one week later →

2. Retention Post-Test

- 5 Code Authoring Tasks
- 5 Code Modification Tasks
- 40 Multiple-Choice Questions

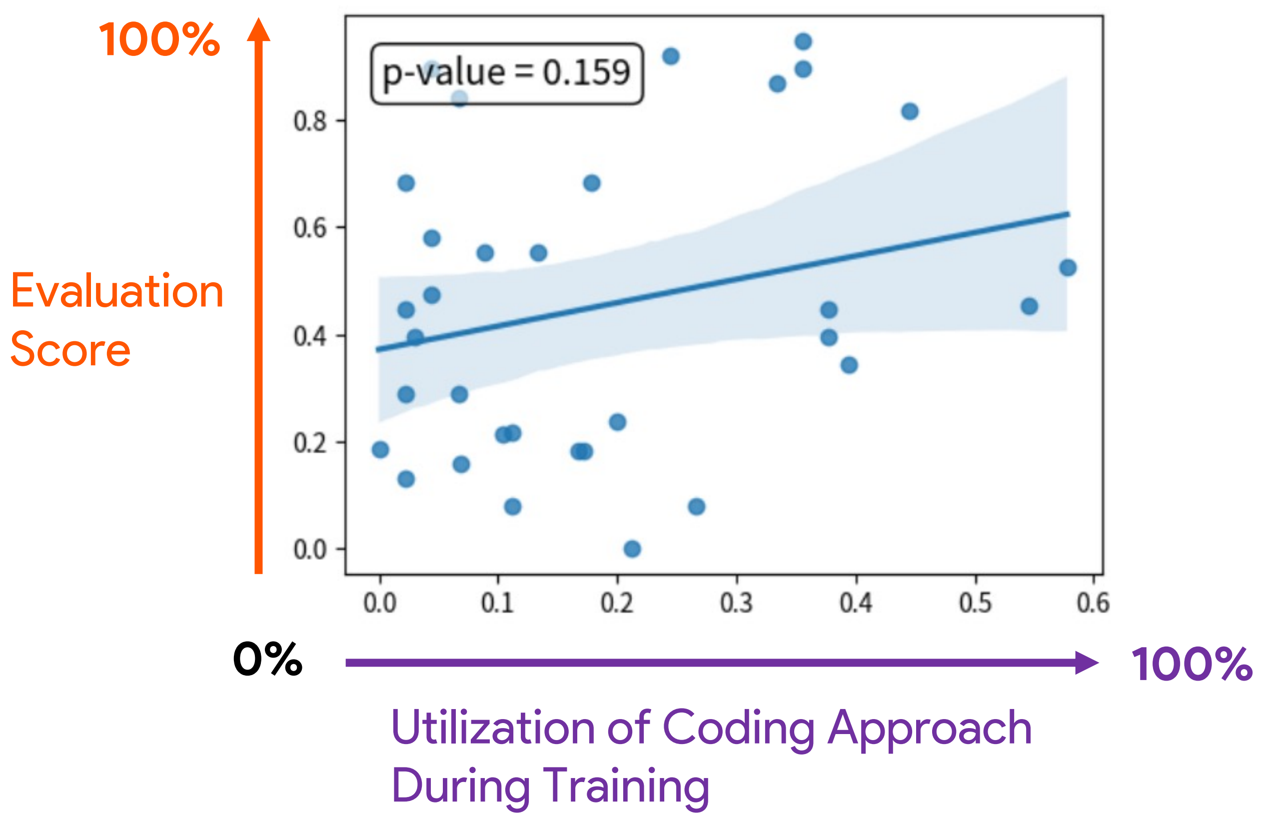
No Python Documentation * No Instructor Hints * No AI Code Generators


Utilization of Each Coding Approach
Each Student during Training:
...
45 Authoring Tasks (7 Training Sessions)
Evaluation Post-Tests
Coding Approaches:
Manual (without Codex)
AI Step-by-Step
Hybrid
AI Single Prompt

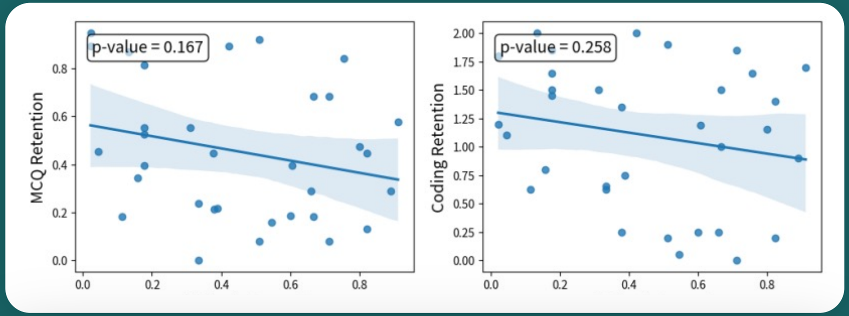
Correlation Analysis:

Utilization of Coding Approach *with* Post-Test Evaluation Score

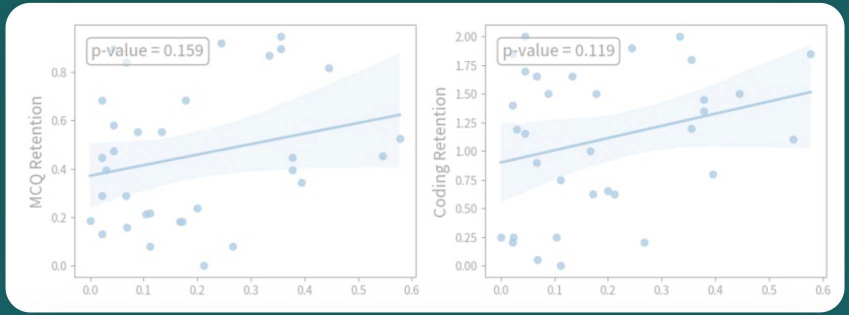
33 dots, each representing a student



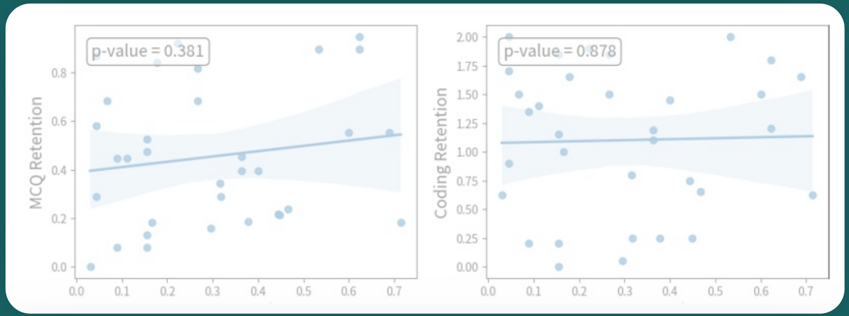
AI Single Prompt



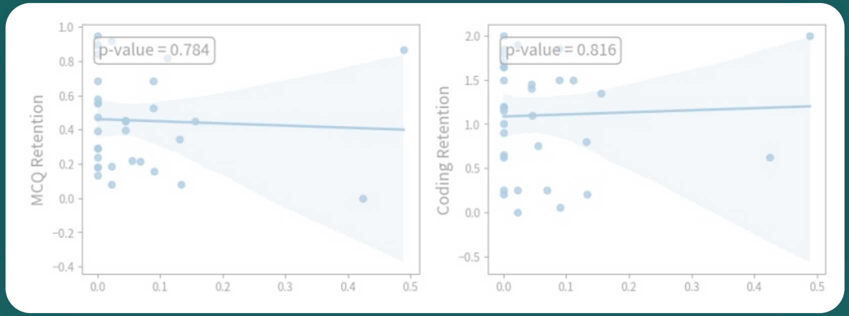
Hybrid



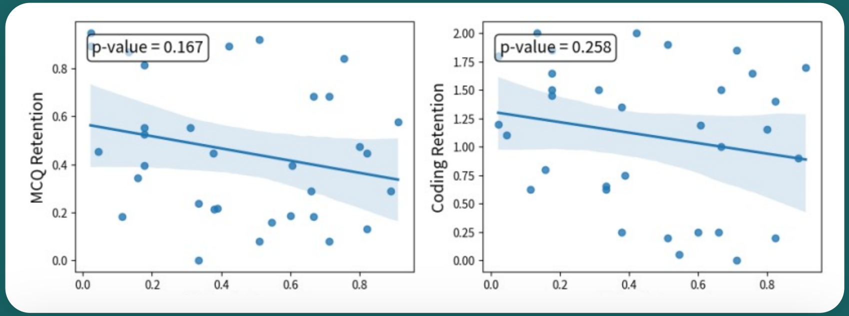
Manual



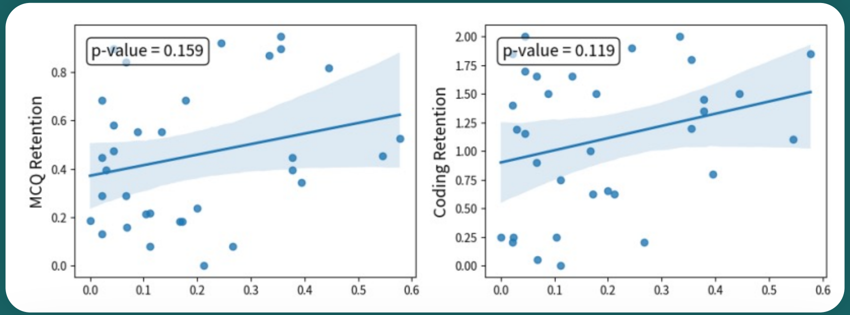
AI Step-by-Step



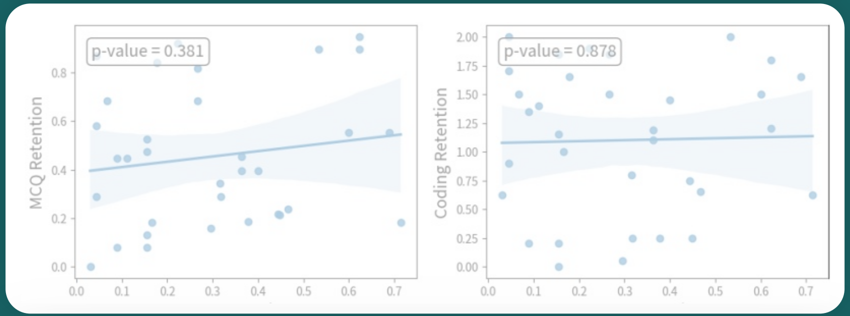
AI Single Prompt



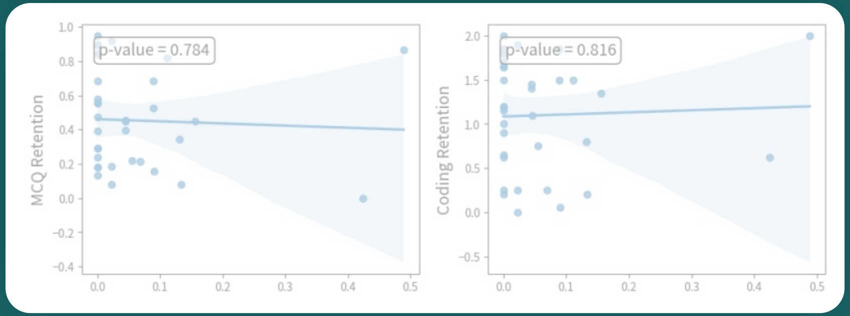
Hybrid



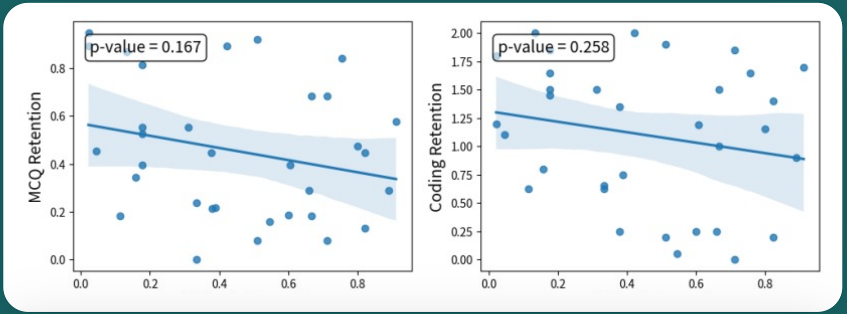
Manual



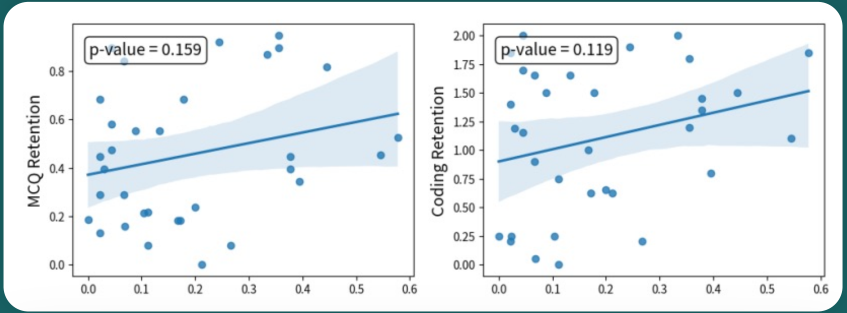
AI Step-by-Step



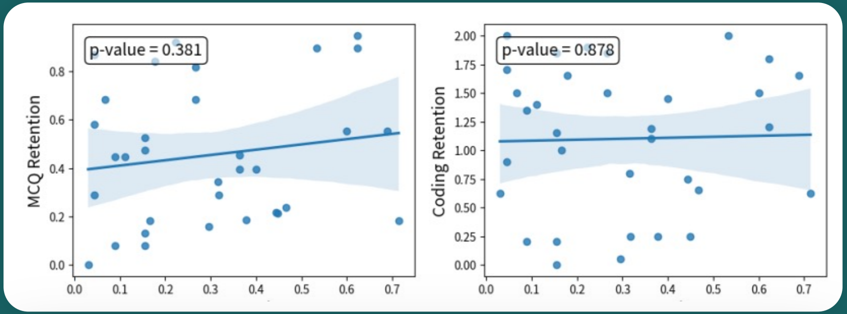
AI Single Prompt



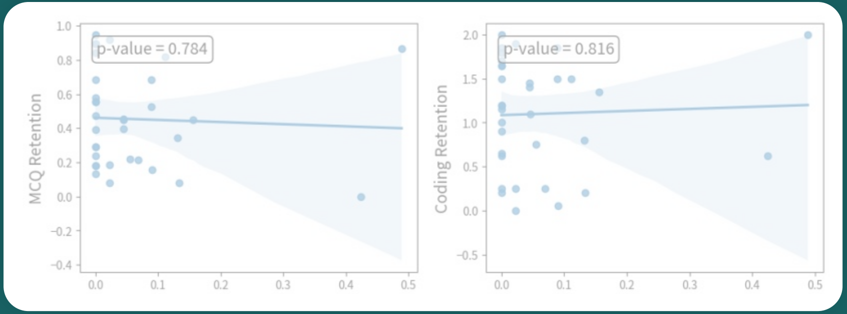
Hybrid



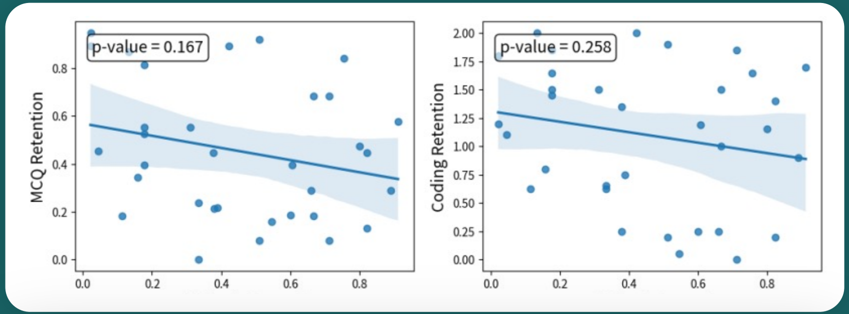
Manual



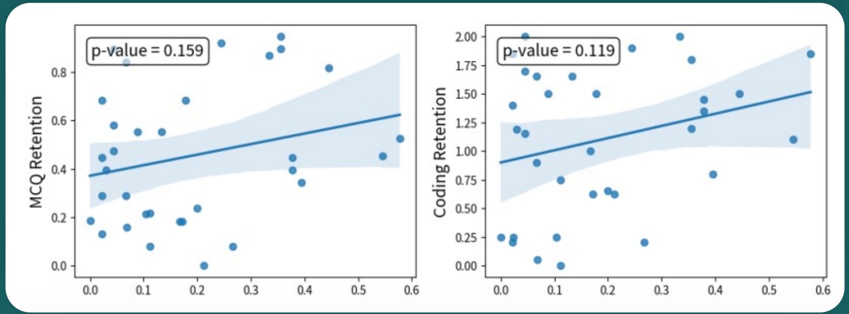
AI Step-by-Step



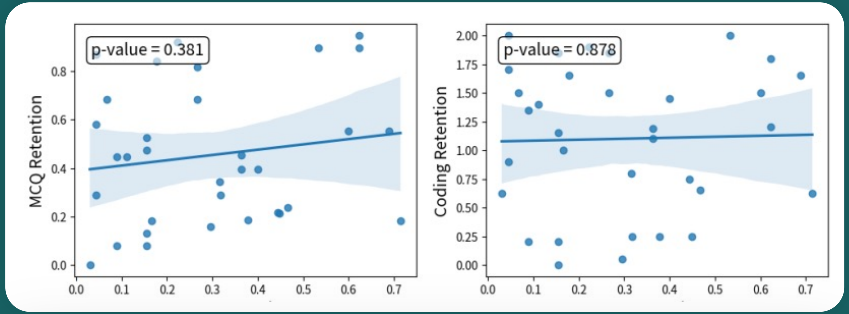
AI Single Prompt



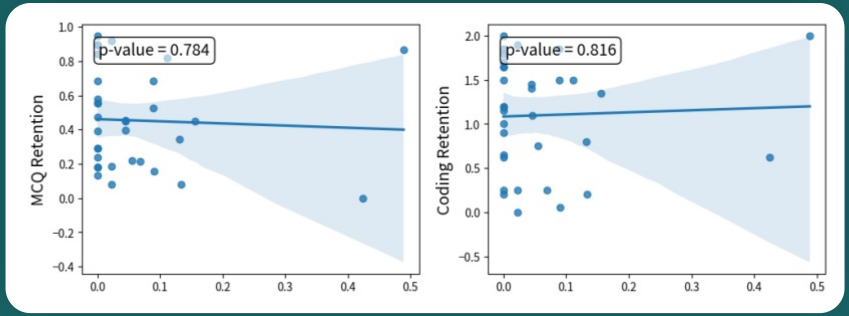
Hybrid



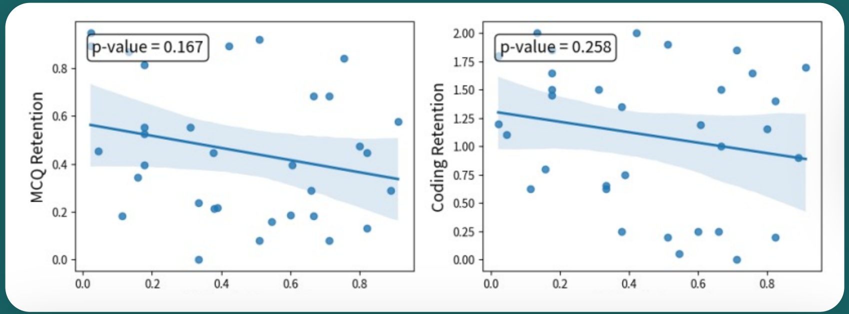
Manual



AI Step-by-Step

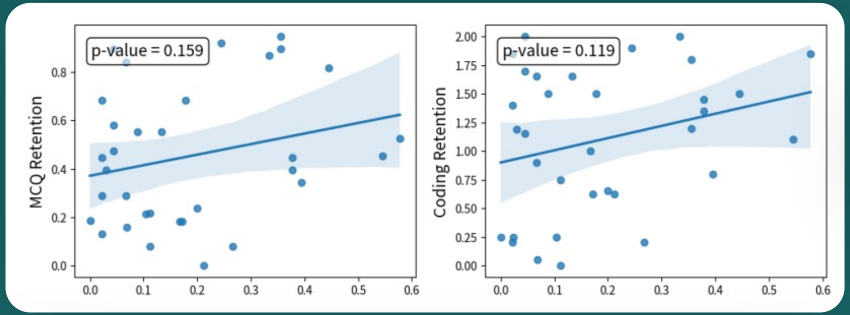


AI Single Prompt



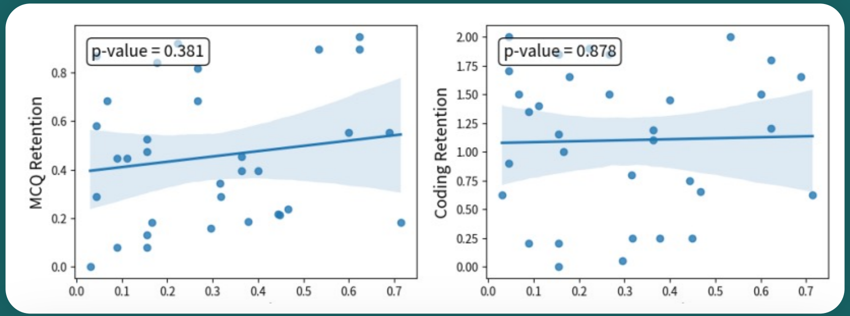
Negative Correlation

Hybrid

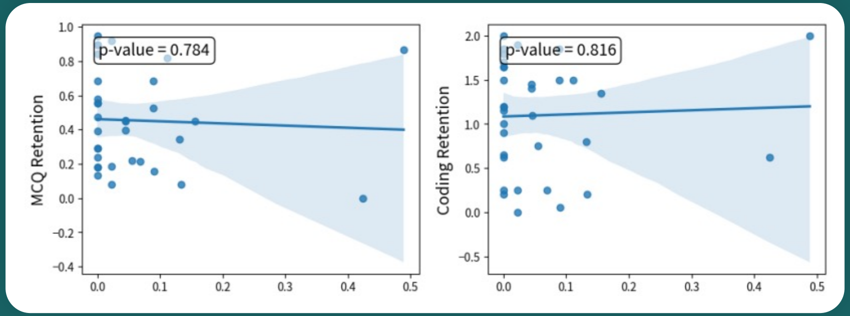


Positive Correlation

Manual



AI Step-by-Step



KEY TAKEAWAYS

Signs of Self-Regulation

- Attempting manual coding before using Codex and using the **Hybrid** AI Coding Approach
- Prompting Codex mainly for syntax
- Actively adding code to verify AI-generated code
- Tinkering with AI-generated code to understand it

Signs of Over-Reliance

- Frequent usage of the AI Single Prompt approach
- Copying the task description and submitting generated code without any editing
- Prompting Codex for code similar to existing code
- Over-trust: submitting code without running

Majeed Kazemitabaar
University of Toronto

Xinying Hou
University of Michigan

Austin Z. Henley
Microsoft Research

Barbara J. Ericson
University of Michigan

David Weintrop
University of Maryland

Tovi Grossman
University of Toronto

Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming (*CHI'23*)
Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, Tovi Grossman

How Novices Use LLM-based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment (*Koli Calling'23*)
Majeed Kazemitabaar, Xinying Hou, Austin Z. Henley, Barbara J. Ericson, David Weintrop, Tovi Grossman

New Paper: CodeAid

To be Presented at CHI 2024

We developed an LLM-powered pedagogical Assistant named CodeAid with five main features that responds to various programming-related questions and help requests.

Unlike unmoderated LLMs, CodeAid produces responses without revealing direct code solutions. Instead, it helps students by producing pseudo-code, suggested fixes and natural language responses.

We deployed CodeAid at a large class of 700 students, interviewed 22 students about their usage, and then interviewed 8 computing educators.

Our results help guide the design of future AI-powered assistants for educational settings.

CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs

Majeed Kazemitabaar
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
majeed@dgp.toronto.edu

Runlong Ye
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
harryye@dgp.toronto.edu

Xiaoning Wang
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
xiaoningwang@dgp.toronto.edu

Austin Z. Henley
Microsoft Research
Redmond, Washington, USA
austinhenley@microsoft.com

Paul Denny
The University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Michelle Craig
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
mcraig@cs.toronto.edu

Tovi Grossman
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada
tovi@dgp.toronto.edu

ABSTRACT

Timely, personalized feedback is essential for students learning programming. LLM-powered tools like ChatGPT offer instant support, but reveal direct answers with code, which may hinder deep conceptual engagement. We developed *CodeAid*, an LLM-powered programming assistant delivering helpful, technically correct responses, without revealing code solutions. CodeAid answers conceptual questions, generates pseudo-code with line-by-line explanations, and annotates student's incorrect code with fix suggestions. We deployed CodeAid in a programming class of 700 students for a 12-week semester. A thematic analysis of 8,000 usages of CodeAid was performed, further enriched by weekly surveys, and 22 student interviews. We then interviewed eight programming educators to gain further insights. Our findings reveal four design considerations for future educational AI assistants: **D1)** exploiting AI's unique benefits; **D2)** simplifying query formulation while promoting cognitive engagement; **D3)** avoiding direct responses while encouraging motivated learning; and **D4)** maintaining transparency and control for students to asses and steer AI responses.

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 the author(s) 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.

CHI '24, May 11–16, 2024, Honolulu, HI, USA

ACM ISBN 979-8-4007-0330-0/24/05…$15.00
https://doi.org/10.1145/3613904.3642773

CCS CONCEPTS

• **Human-centered computing → Interactive systems and tools**; • **Social and professional topics → Computing education**.

KEYWORDS

programming education, intelligent tutoring systems, large language models, educational technology, AI assistants, AI tutoring, generative AI, class deployment, design guidelines

ACM Reference Format:
Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Z. Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24), May 11–16, 2024, Honolulu, HI, USA*. ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3613904.3642773

1 INTRODUCTION

An increasing number of students are learning to program, not just in traditional computer science and engineering degrees, but across a wide range of subject areas [20]. Numerous successful initiatives have been developed to broaden participation in computing, for example, by combining computing majors with disciplines in which there has traditionally been greater gender diversity [7]. However, this surge of interest is putting pressure on resources at many institutions and causing concern amongst administrators and educators [46].

A particularly challenging aspect involves delivering on-the-spot assistance when students need help. Traditional approaches, such as running scheduled office hours in which students can approach instructors and teaching assistants, are often poorly utilized [56].

CHI Conference in Human Factors in Computing
Hawaii, USA, May 2024

Design Goals of Educational AI Assistants

D1: Exploiting Unique Advantages of AI

- **Help-Seeking Choice:** AI assistant vs. Traditional resources
- **AI Tool's Goal:** Productivity-focused vs. Learner-centric
- **Educational Versatility:** General-purpose vs. Course-specific

Key Trade-Off: Broad Scope vs. Unique Advantages

D2: Designing the AI Querying Interface

- **Problem Identification:** Student (reactive) vs. Tool (proactive)
- **Input Format:** Structured vs. Free-form
- **Context Provision:** Manual vs. Automatic

Key Trade-Off: Meta-Cognitive Engagement vs. Ease of Use

D3: Balancing the Directness of AI Responses

- **Directness Level:** Direct vs. Indirect
- **Scaffolding Type:** Hints vs. Leading Questions
- **Agency of Control:** Student vs. Instructor vs. AI

Key Trade-Off: Learning Opportunities vs. Ensuring Progress

D4: Supporting Trust, Transparency and Control

- **Transparency of AI:** Minimal vs. Detailed
- **Validation Point:** After Generation vs. Before Generation
- **AI Steering & Control:** Autonomous vs. User Guided

Key Trade-Off: Ease of Use vs. Transparency and Engagement