Program Synthesis

CS 695 / SWE 699: Programming Tools

Fall 2023

GEORGE MASON UNIVERSITY

Today

- Part 1 (Lecture)(~60 mins)
  - 10 min break!
- Part 2: Tech Talk - GitHub CoPilot (15 mins)
- Part 3: (In-Class Activity)(60 mins)

Logistics

- HW 3 due today
- HW 4 due 11/29

Overview

- What is program synthesis?
- Approaches to program synthesis
- LLM-based program synthesis
- Studies of LLM-based program synthesis

What is program synthesis?

- Developer specifies desired behavior, computer synthesizes a program for this behavior.

Key Questions

- How do you generate a program?
- How do you specify what program behavior is?
- How does a developer check if it works?
- What happens if the program behavior is wrong?

Specification Approaches

- Input / output examples
- Unit tests
- Logical relations between inputs and outputs (specifications)
- User demonstrations
- Keywords describing intent
- Partially complete programs with “holes”
- Natural language description

Synthesis approaches

- Enumerative search
- Genetic programming
- Large language models

Enumerative Search & Genetic Programming

Search space

- Competing goals
  - Expressive: include all programs of interest
  - Restrictive: smaller search space

- Often expressed in terms of what language constructs are or are not allowed
- Examples
  - Expressions only with arithmetic operators
  - Expressions with function invocations & operators
  - Expressions, guarded by one of a specific set of conditionals
  - Loop-free programs with conditionals
  - Expressions with depth a maximum node depth of 4
  - Arbitrary programs

Sumit Gulwani. 2010. Dimensions in program synthesis. *Symposium on Principles and practice of declarative programming* (PPDP ’10), 13-24.

Some methods of reducing search space

- Expressing programs in less expressive domain specific language
  - e.g,. method invocations & conditionals controlling when they exist; control

- Assembling code from existing code snippets
  - Plastic surgery hypothesis: high redundancy in code, so existing code snippets can often be found (and perhaps slightly adapted)

Search techniques

- Brute force
  - Enumerate all programs in the search space
- Version spaces
  - Maintain list of satisfying boolean functions
  - Order from most general to least general
  - Refine as more constraints are added
- Probabilistic inference
  - Estimate distribution elements in search space from data, use to bias search
  - e.g., toString() is far more frequent than xizo(100032)
- Genetic programming
  - Maintain population of programs, use selection, mutation, crossover to evolve
- SAT solvers
  - Represent constraints as logical formula, generate program that satisfies constraint

Sumit Gulwani. 2010. Dimensions in program synthesis. *Symposium on Principles and practice of declarative programming* (PPDP '10), 13-24.

Genetic programming

- One of the oldest approaches, based on genetic algorithms

- Uses analogy with biology
  - DNA —> programs
  - Keep population of programs
  - Select highest scoring programs (e.g., best satisfy constraints) for replication
  - Use crossover & mutation to evolve programs towards better solution

Defect Repair: GenProg

- 1. What is it doing wrong?
  - We take as input a set of negative test cases that characterizes a fault. The input program fails all negative test cases.
- 2. What is it supposed to do?
  - We take as input a set of positive test cases that encode functionality requirements. The input program passes all positive test cases.
- 3. Where should we change it?
  - We favor changing program locations visited when executing the negative test cases and avoid changing program locations visited when executing the positive test cases.
- 4. How should we change it?
  - We insert, delete, and swap program statements and control flow using existing program structure. We favor insertions based on the existing program structure.
- 5. When are we finished?
  - We call the first variant that passes all positive and negative test cases a primary repair. We minimize the differences between it and the original input program to produce a final repair.

Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. 2010. Automatic program repair with evolutionary computation. *Cor*
2010), 109-116.

Example

```
void zunebug(int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear (year)){
      if (days > 366) {
        days -= 366;
        year += 1;
      }
      else {
      }
    }
    else {
      days -= 365;
      year += 1;
    }
  }
  printf("the year is %d\n", year);
}
```

Example

```
if (days > 366)   {
  days -= 366;
  if (days > 366) {   // insert #1
    days -= 366;      // insert #1
    year += 1;        // insert #1
  }                   // insert #1
  year += 1;
}
else {
}
days -= 366;          // insert #2
```

```
if (days > 366)   {
  // days -= 366;          // delete
  // if (days > 366) {     // delete
  //   days -= 366;        // delete
  //   year += 1;          // delete
  // }                     // delete
  year += 1;
}
else {
  days -= 366;        // insert
}
days -= 366;
```

Example

```
void zunebug_repair (int days) {
  int year = 1980;
  while (days > 365) {
    if (isLeapYear (year)) {
      if (days > 366) {
        // days -= 366;  // deleted
        year += 1;
      }
      else {
      }
      days -= 366;       // inserted
    } else {
      days -= 365;
      year += 1;
    }
  }
  printf ("the year is %dn", year);
}
```

Synthesis with Prophet

- **Defect Localization:** The Prophet defect localization algorithm analyzes execution traces of the program running on the test cases in the test suite. The result is a ranked list of target program statements to patch (see Section 3.7). Prophet prioritizes statements that are frequently executed on negative inputs (for which the unpatched program produces incorrect results) and infrequently executed on positive inputs (for which the unpatched program produces correct results).
- **Search Space Generation:** Prophet generates a space of candidate patches, each of which modifies one of the statements identified by the defect localization algorithm.
- **Universal Feature Extraction:** For each candidate patch, Prophet extracts features that summarize relevant patch properties. These features include *program value features*, which capture relationships between how variables and constants are used in the original program and how they are used in the patch, and *modification features*, which capture relationships between the kind of program modification that the patch applies and the kinds of statements that appear near the patched statement in the original program. Prophet converts the extracted features into a binary feature vector.
- **Patch Ranking and Validation:** Prophet uses the learned model and the extracted binary feature vectors to compute a probability score for each patch in the search space of candidate patches. Prophet then sorts the candidates according to their scores and validates the patches against the supplied test suite in that order. It returns an ordered sequence of patches that validate (i.e., produce correct outputs for all test cases in the test suite) as the result of the patch generation process.

Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering* (ICSE '16), 702-713.

Prophet mutation operators

- **Condition Refinement:** Given a target if statement to patch, the system transforms the condition of the if statement by conjoining or disjoining an additional condition to the original if condition. The following two patterns implement the transformation:

  ```
  if (C) { ... } => if (C && P) { ... }
  if (C) { ... } => if (C || P) { ... }
  ```

  Here `if (C) { ... }` is the target statement to patch in the original program. `C` is the original condition that appears in the program. `P` is a new condition produced by a condition synthesis algorithm [18, 20].
- **Condition Introduction:** Given a target statement, the system transforms the program so that the statement executes only if a guard condition is true. The following pattern implements the transformation:

  ```
  S => if (P) S
  ```

  Here `S` is the target statement to patch in the original program and `P` is a new synthesized condition.
- **Conditional Control Flow Introduction:** Before a target statement, the system inserts a new control flow statement (return, break, or goto an existing label) that executes only if a guard condition is true. The following patterns implement the transformation:

  ```
  S => if (P) break; S
  S => if (P) continue; S
  S => if (P) goto L; S
  ```

  Here `S` is the target statement to patch in the original program, `P` is a new synthesized condition, and `L` is an existing label in the procedure containing `S`.
- **Insert Initialization:** Before a target statement, the system inserts a memory initialization statement.
- **Value Replacement:** Given a target statement, replace an expression in the statement with another expression.
- **Copy and Replace:** Given a target statement, the system copies an existing statement to the program point before the target statement and then applies a Value Replacement transformation to the copied statement.

Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering* (ICSE '16), 702-713.

Prophet results

App	LoC	Tests	Defects/ Changes	Plausible					Correct				
				Prophet	SPR	Kali	GenProg	AE	Prophet	SPR	Kali	GenProg	AE
libtiff	77k	78	8/16	5/0	5/0	5/0	3/0	5/0	2,2/0	1,1/0	0/0	0/0	0/0
lighttpd	62k	295	7/2	3/1	3/1	4/1	4/1	3/1	0,0/0	0,0/0	0/0	0/0	0/0
php	1046k	8471	31/13	17/1	16/1	8/0	5/0	7/0	13,10/0	10,9/0	2/0	1/0	2/0
gmp	145k	146	2/0	2/0	2/0	1/0	1/0	1/0	1,1/0	1,1/0	0/0	0/0	0/0
gzip	491k	12	4/1	2/0	2/0	1/0	1/0	2/0	1,1/0	1,0/0	0/0	0/0	0/0
python	407k	35	9/2	5/1	5/1	1/1	0/1	2/1	0,0/0	0,0/0	0/1	0/1	0/1
wireshark	2814k	63	6/1	4/0	4/0	4/0	1/0	4/0	0,0/0	0,0/0	0/0	0/0	0/0
fbc	97k	773	2/1	1/0	1/0	1/0	1/0	1/0	1,1/0	1,0/0	0/0	0/0	0/0
Total			69/36	39/3	38/3	25/2	16/2	25/2	18,15/0	16,11/0	2/1	1/1	2/1

Keyword constraints

```
public List<String> getLines(BufferedReader in) throws Exception {
    List<String> lines = new Vector<String>();
    while (in.ready()) {
        add line
    }
    return lines;
}
```

Ctrl + **Space**

```
public List<String> getLines(BufferedReader in) throws Exception {
    List<String> lines = new Vector<String>();
    while (in.ready()) {
        lines.add(in.readLine());
    }
    return lines;
}
```

- Explore space of expressions, scoring by match of identifiers in expression to provided keywords
- Use in scope variables as leafs in exploration

Greg Little and Robert C. Miller. 2007. Keyword programming in java. *International conference on Automated software engineering* (ASE '07), 84-9

Programming by demonstration

- Program is a set of operations with effects recorded by the user
  - e.g., click a button, enter String in textbox
- User expresses constraints by recording multiple traces
- Goal is to generate program that has same output on demonstrated examples but also work on other similar situations
- Example
  - User selects the first entry from Google search result, pastes that into a form field on another website
  - User demonstrates doing this once (or twice)
  - Want a program that will work for all search results returned by Google

Example: Flashfill

Su 40236.2240260

Challenge: ambiguity

ways to extract 706 from 425-706-7709

- **Second number:** `SubStr2`(ν_1, `NumTok`, 2).
- **2nd last alphanumeric token:** `SubStr2`(ν_1, `AlphNumTok`, −2).
- **Substring between the first hyphen and the last hyphen:** `SubStr`(ν_1, `Pos`(`HyphenTok`, ε, 1), `Pos`(ε, `HyphenTok`, −1)).
- **First number that occurs between hyphen on both ends.**

```
SubStr(ν1, Pos(HyphenTok,
              TokenSeq(NumTok, HyphenTok), 1),
          Pos(TokenSeq(HyphenTok, NumTok),
              HyphenTok, 1) ).
```

- **First number preceded by a number–hyphen sequence.**

```
SubStr(ν1, Pos(TokenSeq(NumTok, HyphenTok),
              NumTok, 1),
          Pos(TokenSeq(NumTok, HyphenTok,
                        NumTok), ε, 1) ).
```

Programming by demonstration

Large language models for programming

GITHUB COPILOT IN 7 MINUTES

Cody EXPERIMENTAL

Chat | Recipes | Debug

The main view of Cody is defined in the `client/cody/webviews/index.tsx` file. This is the entry point for the Cody webview and renders the Chat component, which is the main UI for interacting with Cody.

Cody read the following files to provide an answer:

- client/cody-shared/README.md
- client/cody/index.html
- client/cody/webviews/Chat.tsx
- client/jetbrains/.eslintignore
- client/web/src/codeintel/ReferencesPanel.module.scss
- client/web/src/enterprise/app/AppComingSoonPage.tsx
- client/web/src/enterprise/cody/icon.ts
- client/web/src/setup-wizard/components/remote-repositories-step/RemoteRepositoriesStep.module.scss
- client/web/src/site-admin/SiteAdminConfigurationPage.tsx
- doc/cody/index.md

Cody · 09:49

Can you explain what's going on in Chat.tsx at a high level?

Me · 09:50

Chat.tsx client/cody/webviews/Chat.tsx/Chat

```
        // Submit chat only when input is not empty
        if (formInput !== undefined) {
            vscodeAPI.postMessage({ command: 'submit', text: formInput })
            setHistoryIndex(inputHistory.length + 1)
            setInputHistory([...inputHistory, formInput])
            setInputRows(5)
            setFormInput('')
        }
    }, [formInput, inputHistory, setFormInput, setInputHistory])

    const onChatKeyDown = useCallback(
        (event: React.KeyboardEvent<HTMLDivElement>): void => {
            // Submit input on Enter press (without shift)
            if (event.key === 'Enter' && !event.shiftKey && formInput) {
                event.preventDefault()
                event.stopPropagation()
                onChatSubmit()
            }
            // Loop through input history on up arrow press
            if (event.key === 'ArrowUp' && inputHistory.length) {
                if (formInput === inputHistory[historyIndex] || !formInput) {
                    const newIndex = historyIndex - 1 < 0 ? inputHistory.length - 1 : historyIndex - 1
                    setHistoryIndex(newIndex)
                    setFormInput(inputHistory[newIndex])
                }
            }
        },
        [inputHistory, onChatSubmit, formInput, historyIndex, setFormInput]
    )

    const bubbleClassName = (speaker: string): string => (speaker === 'human' ? 'human' : 'bot')

    useEffect(() => {
        if (transcriptContainerRef.current) {
            // content is being produced.
            // We allow some small threshold for "what is considered not scrol
            // scroll doesn't affect it (ie. if I'm not all the way scrolled
            // I probably still want it to scroll).
```

So that's the post-processing step where we actually

bl/completions | Go 1.19.3 | Beatrix, 2 weeks ago | Ln 99, Col 1 | Spaces: 4 | UTF-8 | LF | TypeScript JSX | Analysis Tools Missing | Go Update Available | Prettier lopes@uci.edu>>> I'm going to teach you how to play a game. In this game, I will prompt you a message and you need to reply something based on that message. I will teach you the rules of the reply by example. Are you ready?

```
Yes, I am ready to learn the game you have described. Please provide examples
and explain the rules of the game.
```

lopes@uci.edu>>> ok, here's the first example.

If I prompt you:

```
{
```

Your reply should be:

```
token(LEFT_PAREN, 1)
```

where token means that you have identified a token in my message, LEFT_PAREN is the type of the token, and 1 is the line number of that token in the prompt. OK?

```
I understand. If you prompt me with a message, my response should be the word
"token", followed by the type of the first token I find in your message
enclosed in parentheses, and the line number where it appears. For example,
if the first token in your message is "(" and it appears on the first line,
my response would be "token(LEFT_PAREN, 1)".
```

http://tagide.com/blog/education/the-end-of-programming-as-we-know-it/

lopes@uci.edu>>> OMG, you're so smart. Yes! You even corrected my mistake in my rule! Of course LEFT_PAREN is (. What do you think we should call the type of the { token?

```
I think the type of the "{" token could be called "LEFT_CURLY_BRACKET"
```

lopes@uci.edu>>> That's a great name! Let's go with that. So here's my prompt now:

```
{
```

What's your reply?

```
token(LEFT_CURLY_BRACKET, 1)
```

lopes@uci.edu>>> YES! Awesome! Here's my next prompt:

```
+
```

What's your reply?

```
token(PLUS, 1)
```

lopes@uci.edu>>> Fantastic! Next prompt:

```
/
```

Your reply?

```
token(FORWARD_SLASH, 1)
```

http://tagide.com/blog/education/the-end-of-programming-as-we-know-it/

lopes@uci.edu>>> ok, let's try.

Prompt:

```
{= == <
>=  !  !=
```

Your reply?

```
token(LEFT_CURLY_BRACKET, 1)
token(EQUAL, 1)
token(EQUAL_EQUAL, 1)
token(LESS_THAN, 2)
token(GREATER_EQUAL, 2)
token(EXCLAMATION, 2)
token(EXCLAMATION_EQUAL, 2)
```

lopes@uci.edu>>> wow. You are amazing! Let me now teach you another rule. When you see a " character, that means that is the beginning of a token of type STRING. That token consists of the sequence of characters following that ", until you see a second closing " character. We have to change our tokens a little bit for strings so that they also store the literal information. For example,

Prompt:

```
"hello world"
```

Your reply should be:

```
token(STRING, 1, hello world)
```

Got it?

http://tagide.com/blog/education/the-end-of-programming-as-we-know-it/

lopes@uci.edu>>> I would love if you could give me a JavaScript implementation of the tokenizer, as complete as possible. Go ahead and create whatever helper functions you need!

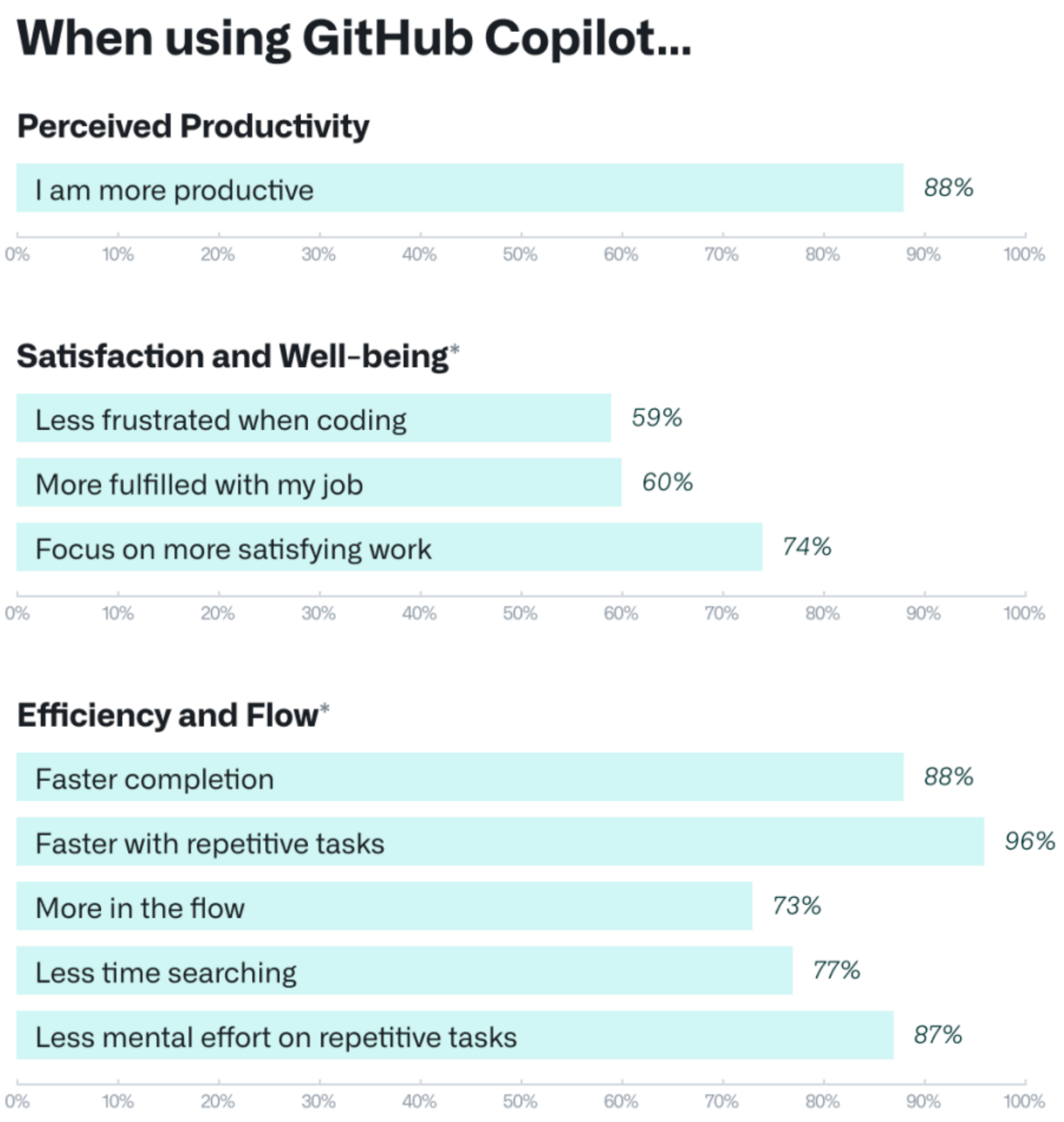
Sure, here is an example of a JavaScript implementation that tokenizes the input string based on the rules you have taught me:

```
const keywords = ["AND", "CLASS", "ELSE", "FALSE", "FUN", "FOR", "IF", "NIL",
"OR", "PRINT", "RETURN", "SUPER", "THIS", "TRUE", "VAR", "WHILE"];

function tokenizer(input) {
    let line = 1;
    let current = 0;
    let tokens = [];
    let inString = false;

    function isDigit(c) {
        return /[0-9]/.test(c);
    }

    function isAlpha(c) {
        return /[a-zA-Z]/.test(c);
    }

    function isAlphaNumeric(c) {
        return isDigit(c) || isAlpha(c);
    }

    while (current < input.length) {
        let char = input[current];
```

http://tagide.com/blog/education/the-end-of-programming-as-we-know-it/

Discussion - Experiences Programming with LLMs

Studies of LLM programming tools



https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

We recruited

95

developers, and split them randomly into two groups.

We gave them the task of writing a web server in JavaScript

45 Used
GitHub Copilot

78%
finished

1 hour, 11 minutes
average to complete the task

71 minutes | **that's 55% less time!**

50 Did not use
GitHub Copilot

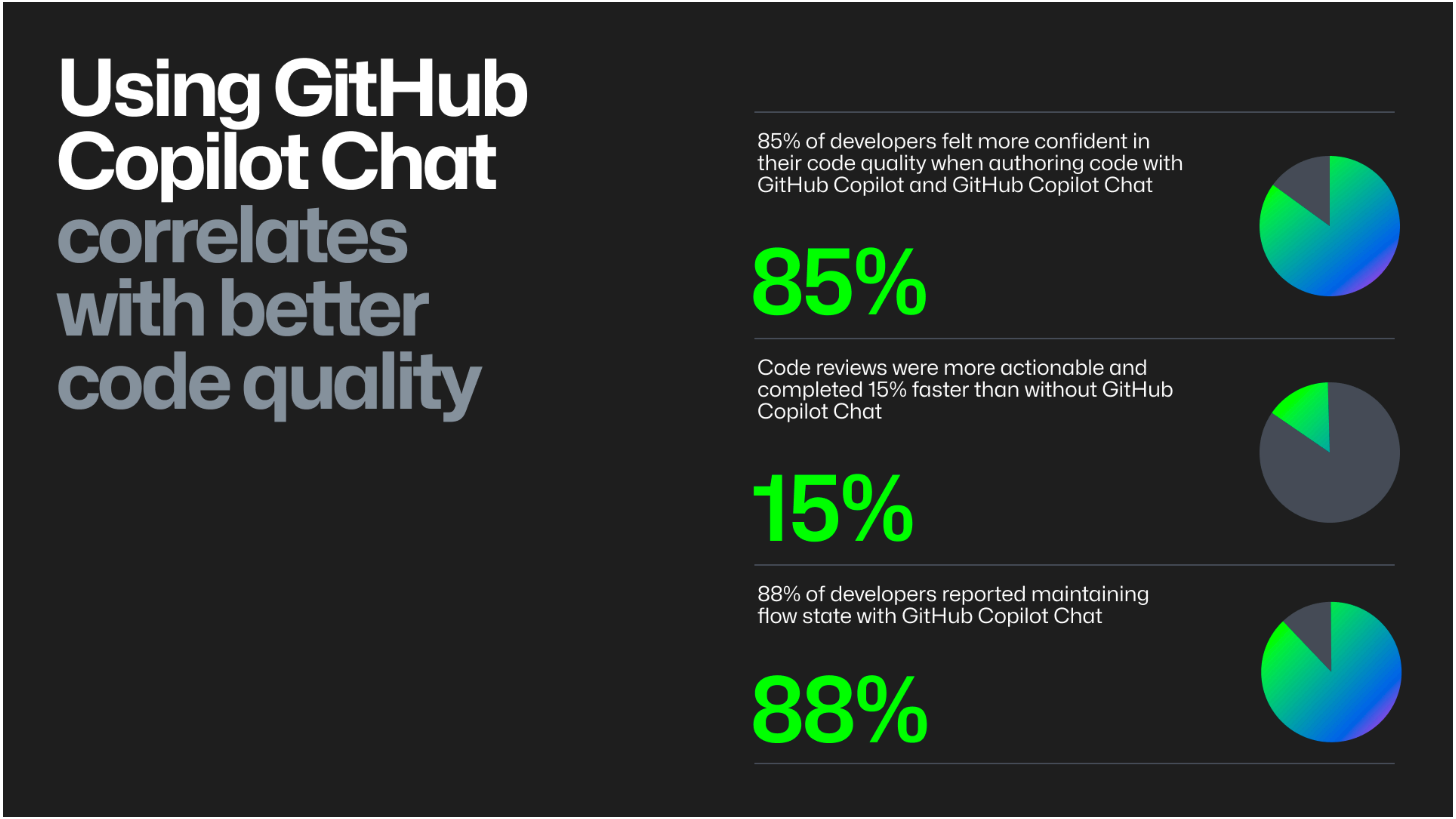
70%
finished

2 hours, 41 minutes
average to complete the task

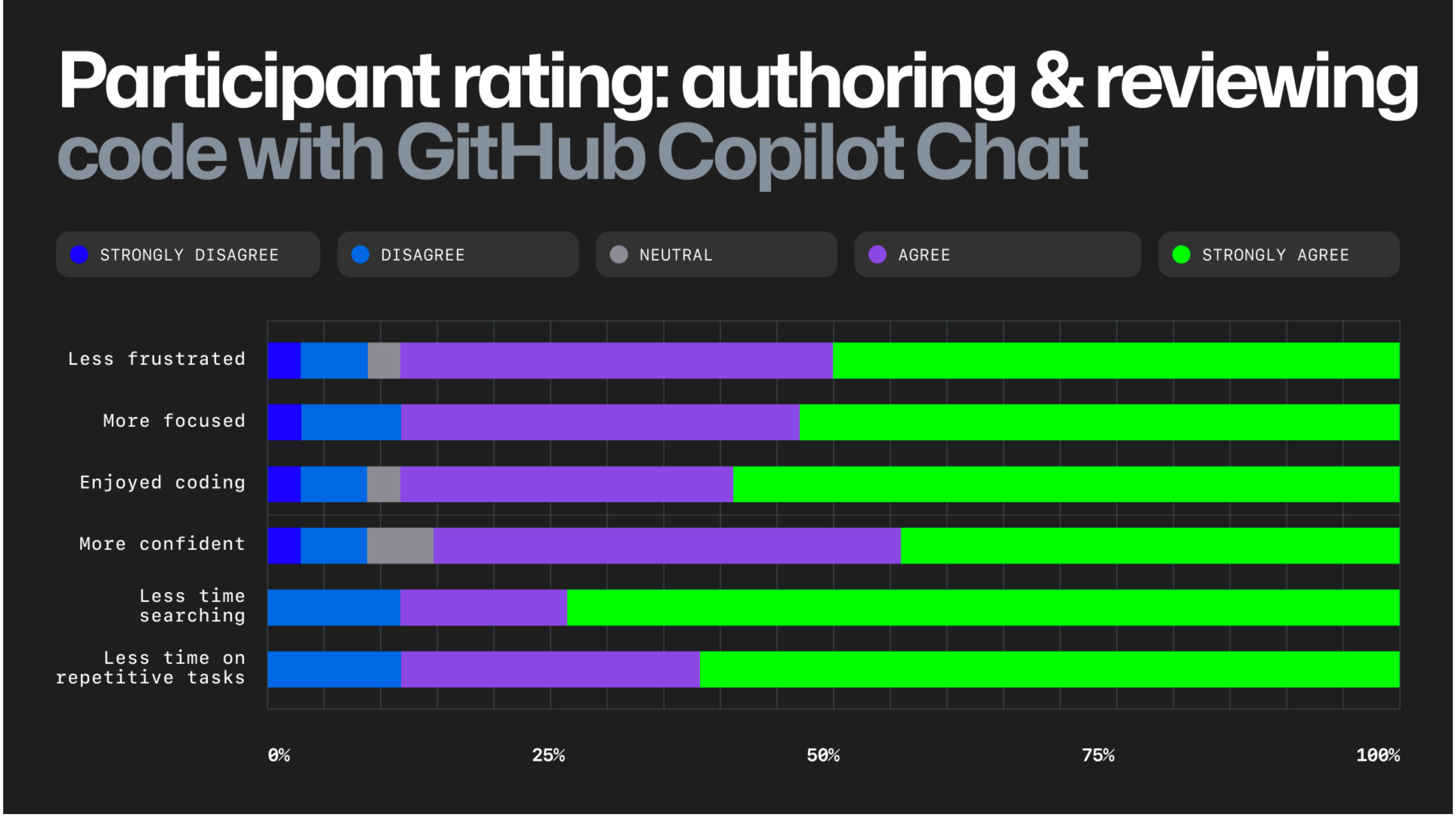
161 minutes

Results are statistically significant (*P*=.0017) and the 95% confidence interval is [21%, 89%]

https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/



https://github.blog/2023-10-10-research-quantifying-github-copilots-impact-on-code-quality/



https://github.blog/2023-10-10-research-quantifying-github-copilots-impact-on-code-quality/

> “
>
> The code was so clean, I could just look at the code, know what was going on, and bring it over fairly easily to my codebase. So, I felt the code quality was very clean and easy to understand. When I was doing the pull request review using Copilot, I saw how good it was at generating proper error handling code.
>
> – Senior Software Engineer at a Fortune 500 company (study participant)

https://github.blog/2023-10-10-research-quantifying-github-copilots-impact-on-code-quality/

Generating Boilerplate

These days not having Copilot is a pretty big productivity hit to me. The other day Copilot somehow stopped offering completions for maybe an hour, and I was pretty shocked to realize how much I've grown to rely on just hitting tab to complete the whole line. (I was writing Go at the time which is on the boilerplatey side among the mainstream languages, so Copilot is particularly effective [...]"

"I use GTP-3 codex [sic] daily when working. It saves me time, helps me explore unfamiliar languages and APIs and generates approaches to solve problems. It can be shockingly good at coding in narrow contexts. It would be a mistake to miss the developments happening in this area"

"[...] for a lot of quick programming questions, I'm finding I don't even need a search engine. I just use Github Copilot. For example, if I wanted to remember how to throw an exception I'd just write that as a comment and let Copilot fill in the syntax. Between that and official docs, don't need a ton else."

"[...] It's changing the way I write code in a way that I can already tell is allowing me to be much lazier than I've previously been about learning various details of languages and libraries. [...]"

"[...] Github Copilot [...] pretty much replaced almost my entire usage of Stack Overflow.[...]"

"[...] GitHub Copilot really shines in rote work: when it can correctly infer what you are about to do, it can and will assist you correctly. It's not able to make big decisions, but in a pinch, it might be able to give hints. [...] If used right, Copilot can give developers a significant velocity boost, especially in greenfield projects where there is lots and lots of boilerplate to write. [...]"

	Task 1 - Easy		Task 2 - Medium		Task 3 - Hard	
	Intellisense	Copilot	Intellisense	Copilot	Intellisense	Copilot
	9:35	1:46	7:48	12:53	13:41	11:08
	3:50	3:57	15:52	16:45	13:43	11:05
	4:49	4:55	16:28	7:26	22:42	4:04
	9:04	6:18	14:16	15:05	13:06	DNF
	5:18	1:18	7:35	13:24	23:13	19:54
	15:54	7:52	12:39	DNF	4:48	DNF
	5:27	3:12	10:47	6:02	DNF	DNF
	2:09	20:12	8:30	DNF	DNF	9:19
Average Time	**7:01**	**6:11**	**11:44**	**11:56**	**13:36**	**11:06**
Overall average time for all tasks combined					**10:23**	**9:18**

Table 1: Individual and average task completion times. Cells with an orange cell background indicate that the participant never succeeded because they were stopped after approximately 20 minutes of trying. DNF implies the participant did not finish on time.

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22). Association for Computing Machinery, New York, NY, USA, Article 332, 1–7. https://doi.org/10.1145/3491101.3519665

Challenges with CoPilot

“I would go with Intellisense for now since it gives me more control over the code I am writing”

"Yes, I got rid of the whole snippet as I didn’t want to conform to the code generated by AI as it may have unwanted bugs.”

- Only trusted for simple tasks, due to the difficulty to understand generated code, fear of unknown bugs, failure to match the coding style

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22). Association for Computing Machinery, New York, NY, USA, Article 332, 1–7. https://doi.org/10.1145/3491101.3519665

CoPilot vs. StackOverflow

"For certain tasks that follow very routine structures, and which I always have to look up on Stack Overflow, a tool like Copilot eliminates a lot of the tedious searching on Google”.

“I’m not fully confident that Copilot will suggest the best solution. By reading Stack Overflow, the helpful thing is that there will always be someone who would just post a better solution, and people will discuss and compare. I feel like that is missing from Copilot.”

“Not exactly sure what this does. I’ll figure it out later”.

"It made debugging the code more difficult as I hadn’t written the code directly and didn’t have an initial intuition about where the bugs might be. Especially with a final bug in my program I really had no idea why it was happening and had to refactor the code.”

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22). Association for Computing Machinery, New York, NY, USA, Article 332, 1–7. https://doi.org/10.1145/3491101.3519665

Giving up on CoPilot

“I think getting rid of the whole code is easier than reading the code and making the changes.”

CoPilot generated a regex that was hard to fix. Instead of finding a different approach, participant spent 20 minutes unsuccessfully trying to fix it

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22). Association for Computing Machinery, New York, NY, USA, Article 332, 1–7. https://doi.org/10.1145/3491101.3519665

Acceleration vs. Exploration

- Acceleration - use CoPilot to complete code faster

```python
# rules are formatted like:
# AB => C
def parse_input(filename):
    with open(filename) as f:
        template, rules = f.read().split("\n\n")
        for rule in rules:
            rule_parts = rule.split(" => ")
```

- Exploration - find starting points, explore options

```python
You, now | 1 author (You)
import matplotlib
import matplotlib.pyplot as plt

def read_first_digits_from_file(filename):
    with open(filename) as file:
        data = file.read().splitlines()
    return [int(line[0]) for line in data]

fib_first_digits = read_first_digits_from_file("fib.
inverse_first_digits = read_first_digits_from_file('

# Plot the first digits of the Fibonacci
# sequence as a histogram        You, now • Uncommitte
```

```
=======

Accept Solution
# Plot the first digits of the Fibonacci sequence as
plt.hist(fib_first_digits, bins=range(0, 10))
plt.title("Fibonacci sequence")
plt.xlabel("First digit")
plt.ylabel("Number of occurrences")
plt.savefig("fib.png")

=======

Accept Solution
# Plot the first digits of the Fibonacci sequence as
plt.hist(fib_first_digits, bins=range(0, 10))
plt.title("Fibonacci sequence")
plt.xlabel("First digit")
plt.ylabel("Number of occurrences")
plt.show()

=======

Accept Solution
# Plot the first digits of the Fibonacci sequence as
plt.hist(fib_first_digits, bins=10, range=(0, 10))
plt.title("Fibonacci sequence")
plt.xlabel("First digit")
plt.ylabel("Number of occurrences")
plt.savefig("fib.png")
```

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. Proc. ACM Program. Lang. 7, OOPSLA1, Article 78 (April 2023), 27 pages. https://doi.org/10.1145/3586030

Acceleration

- Requires first decomposing problem into subproblems
- Accept end of line suggestions for small logical units - function calls or argument completions
- Long suggestions break flow and are dismissed
- Validated by checking for presence of key function calls or variable names developer expects to see
- Otherwise, rejected

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. Proc. ACM Program. Lang. 7, OOPSLA1, Article 78 (April 2023), 27 pages. https://doi.org/10.1145/3586030

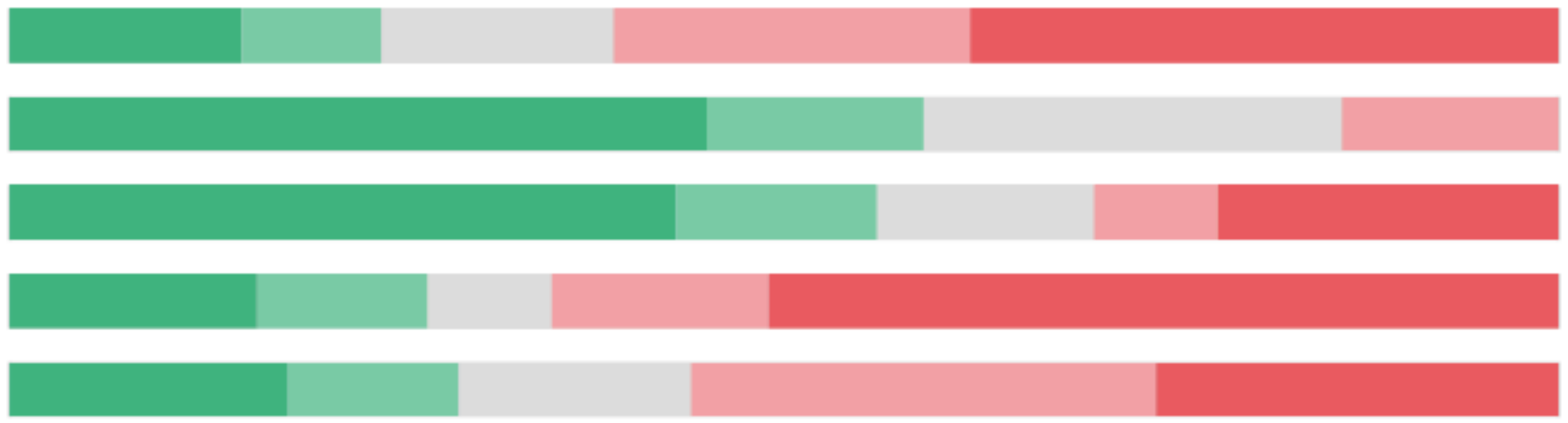
Exploration

- Requires developer to first trust the model
- Prompt with comments rather than code
- Rewrite when suggestions don't match expectations
- Frequently remove comments after completing task
- Explore multiple suggestions through multi selection pane, taking parts or combining parts
- Give more confidence when repeats variations of a similar approach
- Carefully examined and validate suggestions with code examination, testing, documentation
- Willing to accept and edit

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. Proc. ACM Program. Lang. 7, OOPSLA1, Article 78 (April 2023), 27 pages. https://doi.org/10.1145/3586030

Table 1: Participants' self-reported usage of popular AI programming assistants. An asterisk (*) denotes a write-in suggestion, which has limited information on its usage distribution. Percentages in italics on the chart (*N %*) represent the percent of the distribution that reported "Always"/"Often" (left) and "Rarely"/"Tried but gave up" (right).

Tool	# users	Med. % code written	Usage distribution (Always/Often)	Usage distribution (Rarely/Tried but gave up)
Amazon CodeWhisperer	50	5%	*24%*	*61%*
ChatGPT*	25	20%	*59%*	*14%*
GitHub Copilot	306	30.5%	*46%*	*30%*
TabNine	118	20%	*27%*	*66%*
Organization-specific code generation tool trained on proprietary code	54	37%	*29%*	*56%*



Always (1+ times daily) | Often (once daily) | Sometimes (weekly) | Rarely (monthly) | Tried but gave up

Jenny T. Liang, Chenyang Yang, Brad A. Myers. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. International Conference on Software Engineering (ICSE), 2024

Table 2: Participants' motivations for using and not using AI programming assistants.

	Motivation	Distribution	
	A. For using		
M1	To have an autocomplete or reduce the amount of keystrokes I make.	86%	6.2%
M2	To finish my programming tasks faster.	76%	12%
M3	To skip needing to go online to find specific code snippets, programming syntax, or API calls I'm aware of, but can't remember.	68%	14%
M4	To discover potential ways or starting points to write a solution to a problem I'm facing.	50%	24%
M5	To find an edge case for my code I haven't considered.	36%	44%
	B. For not using		
M6	Code generation tools write code that doesn't meet functional or non-functional (e.g., security, performance) requirements that I need.	54%	34%
M7	It's hard to control code generation tools to get code that I want.	48%	36%
M8	I spend too much time debugging or modifying code written by code generation tools.	38%	45%
M9	I don't think code generation tools provide helpful suggestions.	34%	46%
M10	I don't want to use a tool that has access to my code.	30%	51%
M11	I write and use proprietary code that code generation tools haven't seen before and don't generate.	28%	59%
M12	To prevent potential intellectual property infringement.	26%	66%
M13	I find the tool's suggestions too distracting.	26%	51%
M14	I don't understand the code written by code generation tools.	16%	76%
M15	I don't want to use open-source code.	10%	89%

Very important / Important / Moderately important / Slightly important / Not important at all

Jenny T. Liang, Chenyang Yang, Brad A. Myers. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. International Conference on Software Engineering (ICSE), 2024

Successful use cases

- Repetitive code - boilerplate, endpoints
- Code with simple logic - independent util functions, sorting algorithms, small functions
- Autocomplete
- Quality assurance - log messages, test cases
- Proof of concept - only had fuzzy idea how to approach
- Learning - new programming languages or libraries
- Recalling - syntax of languages and API methods

Jenny T. Liang, Chenyang Yang, Brad A. Myers. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. International Conference on Software Engineering (ICSE), 2024

User Input Strategies

- Clear and explicit explanations in code
- Adding code for additional context
- Breaking down instructions of desired behavior into step by step parts
- Prompt engineering - simpler sentences, different language

Jenny T. Liang, Chenyang Yang, Brad A. Myers. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. International Conference on Software Engineering (ICSE), 2024

Table 3: How frequently participants report usability issues occurring while using AI programming assistants.

	Situation	Distribution (left)	Distribution (right)
	A. Usability issues		
S1	I don't know what part of my code or comments the code generation tool is using to make suggestions.	30%	48%
S2	I give up on incorporating the code created by a code generation tool and write the code myself.	28%	35%
S3	I have trouble controlling the tool to generate code that I find useful.	26%	48%
S4	I find the code generation tool's suggestions too distracting.	23%	44%
S5	I have trouble evaluating the correctness of the generated code.	23%	52%
S6	I have difficulty expressing my intent or requirements through natural language to the tool.	22%	36%
S7	I find it hard to debug or fix errors in the code from code generation tools.	17%	61%
S8	I rely on code generation tools too much to write code for me.	15%	67%
S9	I have trouble understanding the code created by a code generation tool.	5.6%	45%
	B. Reasons for not understanding code output		
S10	The generated code uses APIs or methods I don't know.	25%	43%
S11	The generated code is too long to read quickly.	23%	45%
S12	The generated code contains too many control structures (e.g., loops, if-else statements).	19%	44%
	C. Methods of evaluating code output		
S13	Quickly checking the generated code for specific keywords or logic structures	74%	10%
S14	Compilers, type checkers, in-IDE syntax checkers, or linters	71%	14%
S15	Executing the generated code	69%	14%
S16	Examining details of the generated code's logic in depth	64%	15%
S17	Consulting API documentation	38%	28%
	D. Methods of modifying code output		
S18	When a code generation tool outputs something I don't want, I'm able to modify it to something I want.	63%	12%
S19	I successfully incorporate the code created by a code generation tool by changing the generated code.	62%	9.1%
S20	I use the code created by a code generation tool as-is.	44%	24%
S21	I successfully incorporate the code created by a code generation tool by changing the code or comments around it and regenerating a new suggestion.	40%	30%
	E. Reasons for giving up on code output		
S22	The generated code doesn't perform the action I want it to do.	43%	22%
S23	The generated code doesn't meet functional or non-functional (e.g., security, performance) requirements that I need.	34%	28%
S24	The generated code's style doesn't match my project's.	22%	48%
S25	The generated code contains too many defects.	21%	42%
S26	The generated code uses an API I know, but don't want to use.	17%	55%
S27	I don't understand the generated code well enough to use it.	12%	69%
S28	The generated code is too complicated.	10%	68%
S29	The generated code uses an API I don't know.	10%	59%

Always   Often   Sometimes   Rarely   Never

Jenny T. Liang, Chenyang Yang, Brad A. Myers. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. International Conference on Software Engineering (ICSE), 2024

Improving interactions

- User feedback - correct outputted code
- Better under code context from other files
- Flip between acceleration and exploration mode
- Chat to refine code behavior
- Ensure code runs
- More explanation, links to documentation
- More suggestions
- Account for non-functional requirements such as performance

Jenny T. Liang, Chenyang Yang, Brad A. Myers. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. International Conference on Software Engineering (ICSE), 2024

10 min break

Tech Talk: GitHub CoPilot

In-Class Activity

- In groups of 2, try out GitHub CoPilot
  - https://github.com/features/copilot
  - Setup the free trial
  - Setup VS Code
  - Build a simple front end web app game
    - Build tetris
  - Reflect on your experiences with CoPilot
    - What were you able to accomplish (totally ok if didn't finish)
    - What worked well
    - What didn't work well

- Submission
  - Submit (1) pdf or doc with reflection and (2) zip file with source code through Blackboard. 1 submission per group. Due 7pm today.