Interacting with Code Generators

02/29/2023
CMU Neural Code Generation

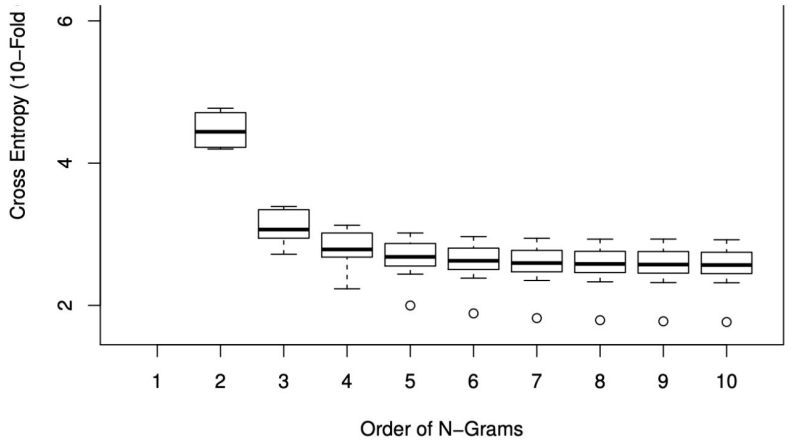
In 2012...

Natural languages like English are rich, complex, and powerful. We begin with the conjecture that most software is also natural, in the sense that it is created by humans at work, with all the attendant constraints and limitations—and thus, like natural language, it is also likely to be repetitive and predictable. We then proceed to ask whether a) code can be usefully modeled by statistical language models and b) such models can be leveraged to support software engineers.

"On the Naturalness of Software"
[Hindle et al., ICSE 2012. Most Influential Paper 2022]

In 2012...

- N-gram models trained on 14 million tokens of code
- Substantial improvements to Eclipse IDE's auto-complete
- Key differences:
  - 2-4 orders of magnitude less data than modern LLMs
  - Count-based n-gram vs. transformer



"On the Naturalness of Software"
[Hindle et al., ICSE 2012. Most Influential Paper 2022]

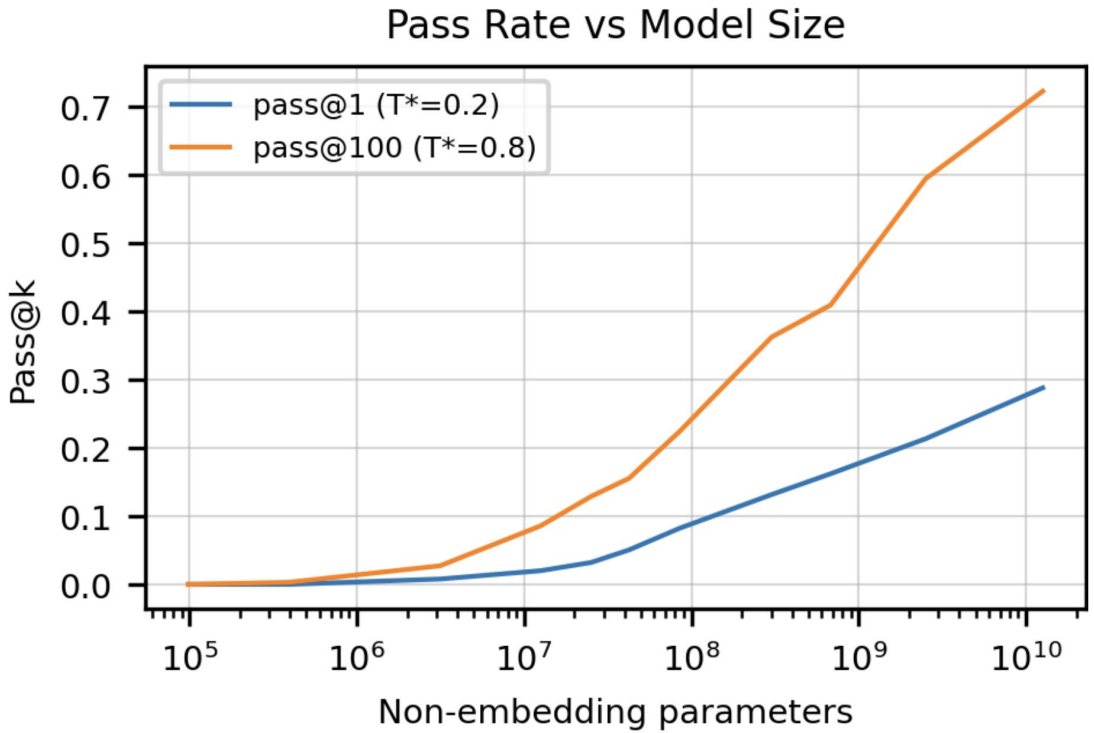
... and now

OpenAI's Codex [Evaluating Large Language Models Trained on Code. Chen et al., 2021]

- Transformer LMs trained on ~100 **billion** tokens of Python code from GitHub
- Typical benchmark: Python function generation:

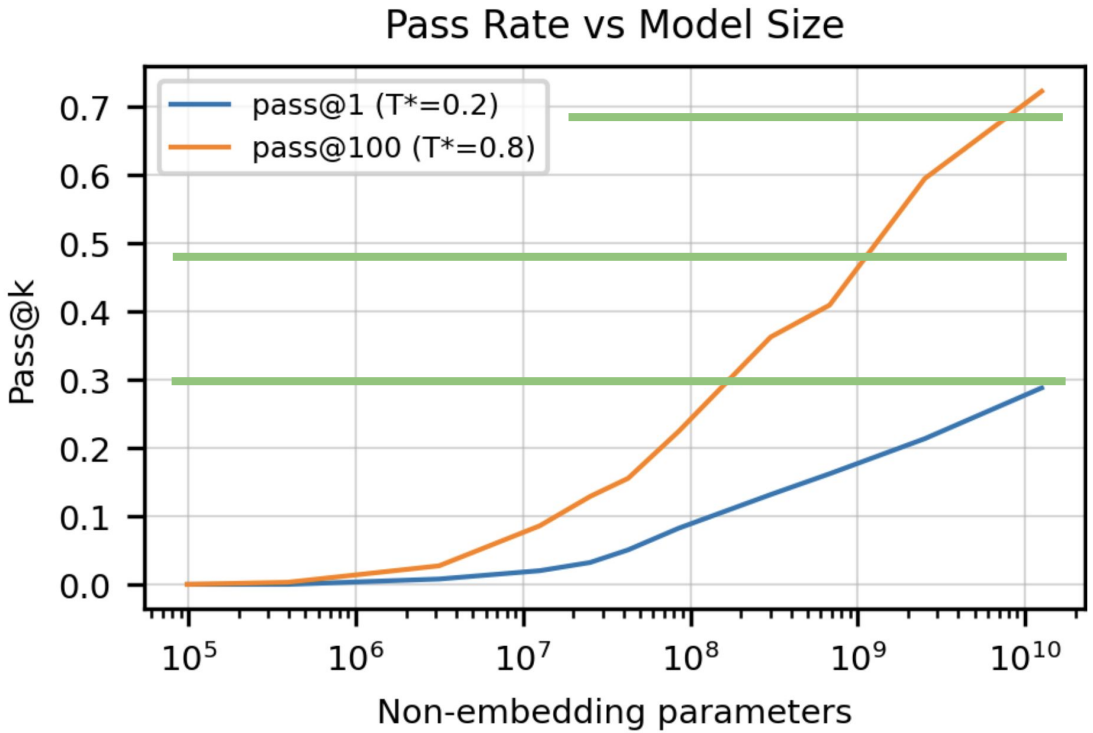
```
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    Examples
    solution([5, 8, 7, 1]) ==>12
    solution([3, 3, 3, 3, 3]) ==>9
    solution([30, 13, 24, 321]) ==>0
    """
    return sum(lst[i] for i in range(0,len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

... and now



OpenAI's Codex [Evaluating Large Language Models Trained on Code. Chen et al., 2021]

... and now



GPT-4

codex (code-davinci-002), and GPT-3.5

Copilot (code-cushman-001)

With more than 1.2 million developers in our technical preview over the last 12 months, people who started using GitHub Copilot quickly told us it became an indispensable part of their daily workflows. In files where it's enabled, nearly 40% of code is being written by GitHub Copilot in popular coding languages, like Python—and we expect that to increase. That's creating more time and space for developers to focus on solving bigger problems and building even better software.

And now, you can put the power of GitHub Copilot to work in your preferred environment with a free trial.

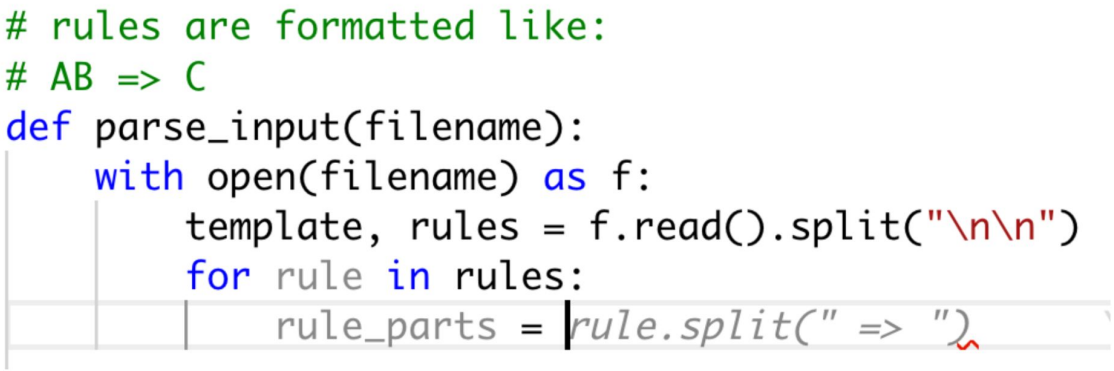
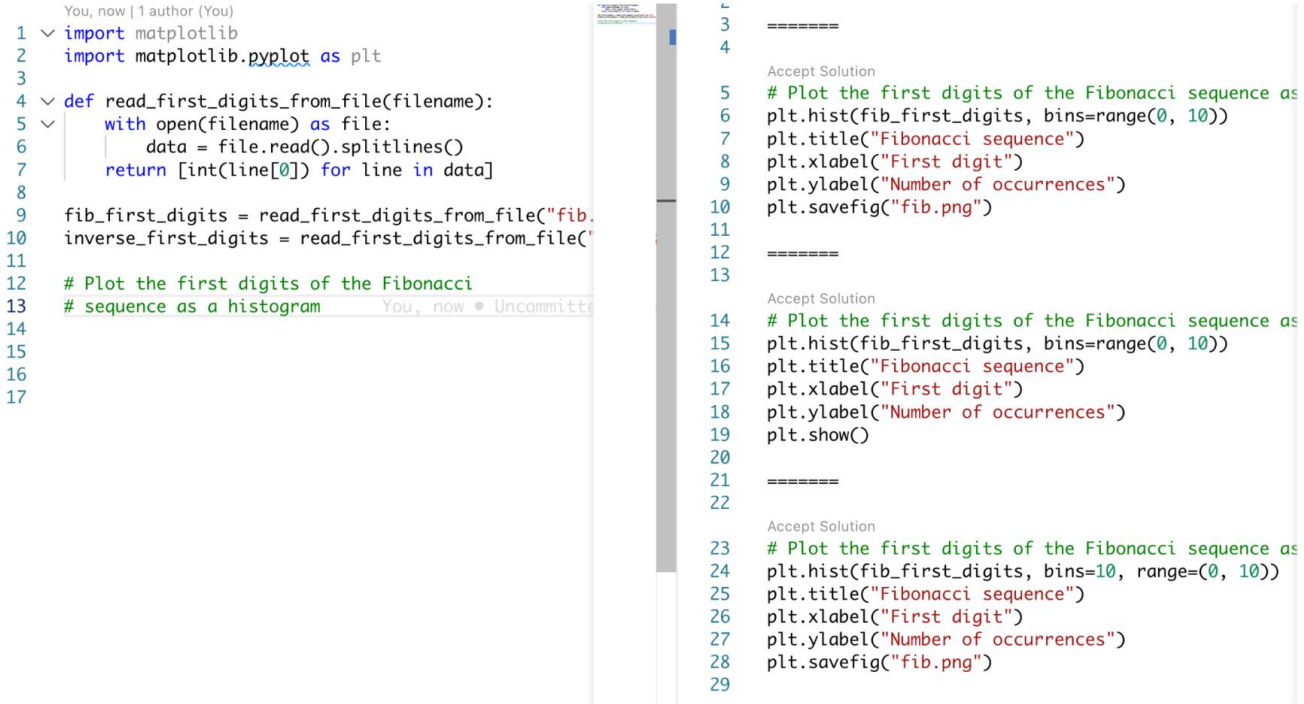


Fig. 1. Copilot's end-of-line suggestion appears at the cursor without explicit invocation. The programmer can press <tab> to accept it.

[Grounded Copilot, Barke et al OOPSLA 2023]



```
You, now | 1 author (You)
1  import matplotlib
2  import matplotlib.pyplot as plt
3
4  def read_first_digits_from_file(filename):
5      with open(filename) as file:
6          data = file.read().splitlines()
7      return [int(line[0]) for line in data]
8
9  fib_first_digits = read_first_digits_from_file("fib.
10 inverse_first_digits = read_first_digits_from_file("
11
12 # Plot the first digits of the Fibonacci
13 # sequence as a histogram      You, now • Uncommitte
14
15
16
17
```

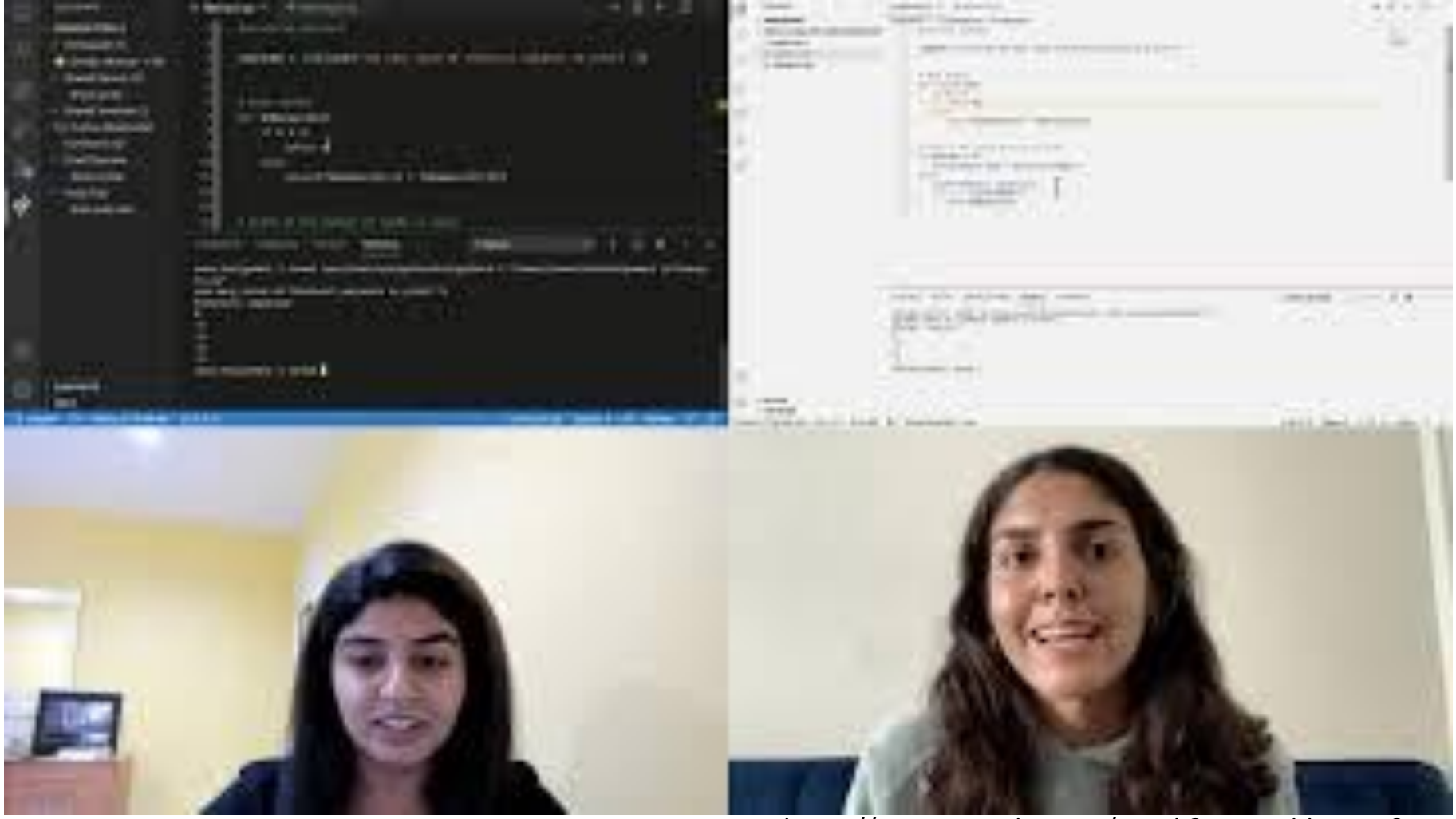
```
3  =======
4
   Accept Solution
5  # Plot the first digits of the Fibonacci sequence as
6  plt.hist(fib_first_digits, bins=range(0, 10))
7  plt.title("Fibonacci sequence")
8  plt.xlabel("First digit")
9  plt.ylabel("Number of occurrences")
10 plt.savefig("fib.png")
11
12 =======
13
   Accept Solution
14 # Plot the first digits of the Fibonacci sequence as
15 plt.hist(fib_first_digits, bins=range(0, 10))
16 plt.title("Fibonacci sequence")
17 plt.xlabel("First digit")
18 plt.ylabel("Number of occurrences")
19 plt.show()
20
21 =======
22
   Accept Solution
23 # Plot the first digits of the Fibonacci sequence as
24 plt.hist(fib_first_digits, bins=10, range=(0, 10))
25 plt.title("Fibonacci sequence")
26 plt.xlabel("First digit")
27 plt.ylabel("Number of occurrences")
28 plt.savefig("fib.png")
29
```

Fig. 2. The user writes an explicit comment prompt (lines 12–13 on the left) and invokes Copilot's multi-suggestion pane by pressing <ctrl> + <enter>. The pane, shown on the right, displays up to 10 unique suggestions, which reflect slightly different ways to make a histogram with `matplotlib`.

[Grounded Copilot, Barke et al OOPSLA 2023]

Programming as Communication

Natural languages like English are rich, complex, and powerful. We begin with the conjecture that most software... is created by humans at work, ~~with all the attendant constraints and limitations~~—
communicating with the compiler, other developers, and themselves,
and thus, like natural language,
~~it is also likely to be repetitive and predictable~~.
writing software is a form of contextual and interactive communication.
We then proceed to ask whether a) code can be usefully modeled by statistical language models and b) such models can be leveraged to support software engineers.



https://www.youtube.com/watch?v=A2ceblXTBBc&t=180s

How Do You Program?

Do you use StackOverflow?

Do you write unit tests?

Do you pair program?

Do you use Copilot? What's been your experience?

What would you want in an AI-assisted programming system?

What is it like to program with artificial intelligence?

Advait Sarkar, Andrew D. Gordon, Carina Negreanu,
Christian Poelitz, Sruti Srinivasa Ragavan, Ben Zorn

PPIG (Psychology of Programming Interest Group) '22

https://arxiv.org/abs/2208.06213

Prior Conceptualizations of AI Programming

- Form and function
  - Direct manipulation (WYSIWYG) works for text/images, but hard for programs (which have multiple possible execution paths/possible futures)
  - Programming by demonstration or examples. Macro recording, FlashFill, program induction
  - Declarative programming: Prolog and descendants. Formal specifications
- Activities (Green 1989)
  - Authoring (write code, or examples, or specifications)
  - Transcription (copy code with changes)
  - Modification (refactor code)
- ML + PL + Software Engineering
  - Treat software as modellable data, often using NLP

Survey of Usability Studies

- Vaithilingam et al. 2022
  - No effects on task time; more failures with Copilot (wild goose chases)
  - But people preferred Copilot anyway! Good starting point

- Ziegler et al. 2022 (we will look at this next)
  - Acceptance rate correlates with perceived productivity
  - Highest acceptance rate on weekends & outside of working hours

Productivity Assessment of Neural Code Completion

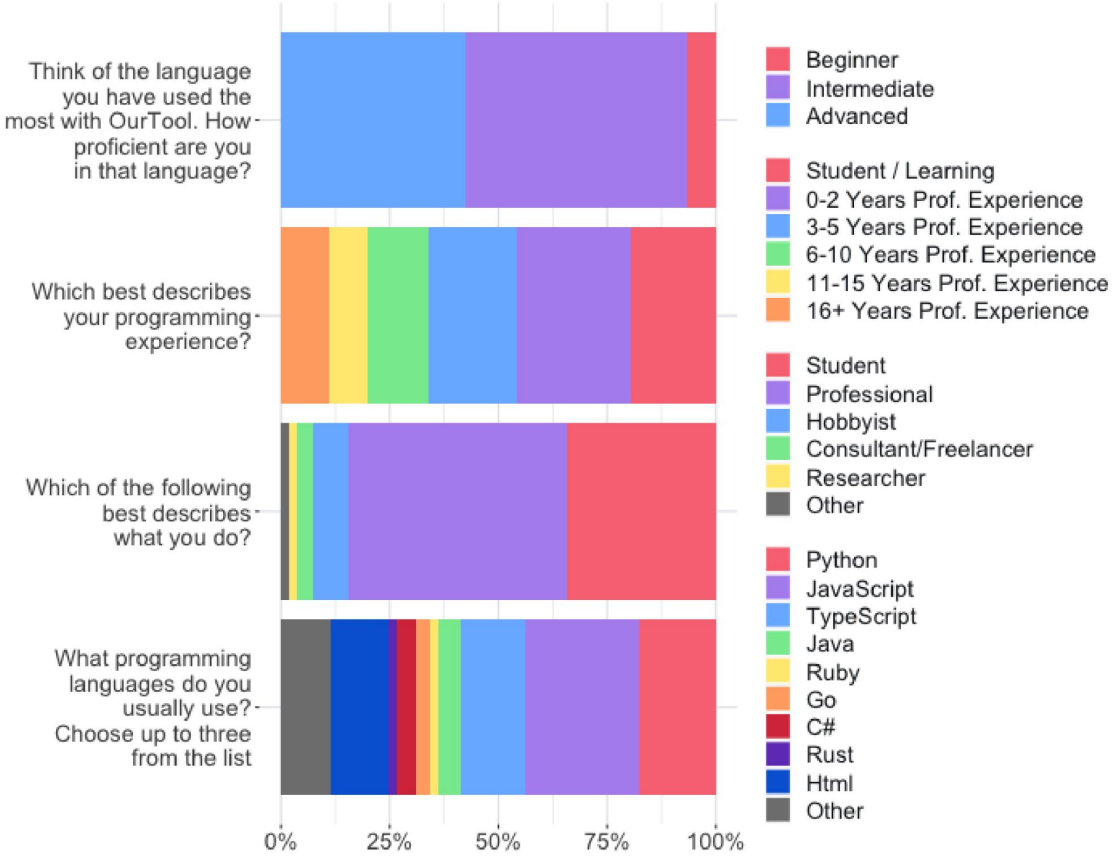
Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin, Edward Aftandilian

MAPS (Intl. Symposium on Machine Programming) '22

https://arxiv.org/abs/2205.06537

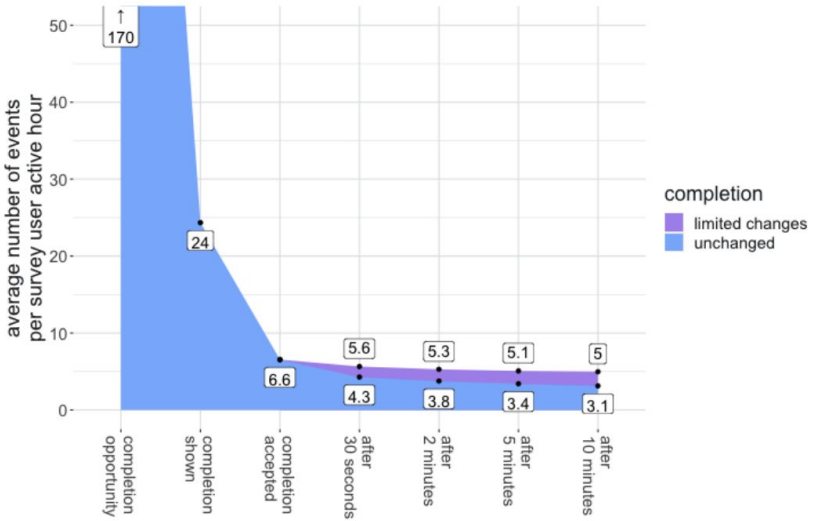
Productivity Assessment of Neural Code Completion (Ziegler '22)

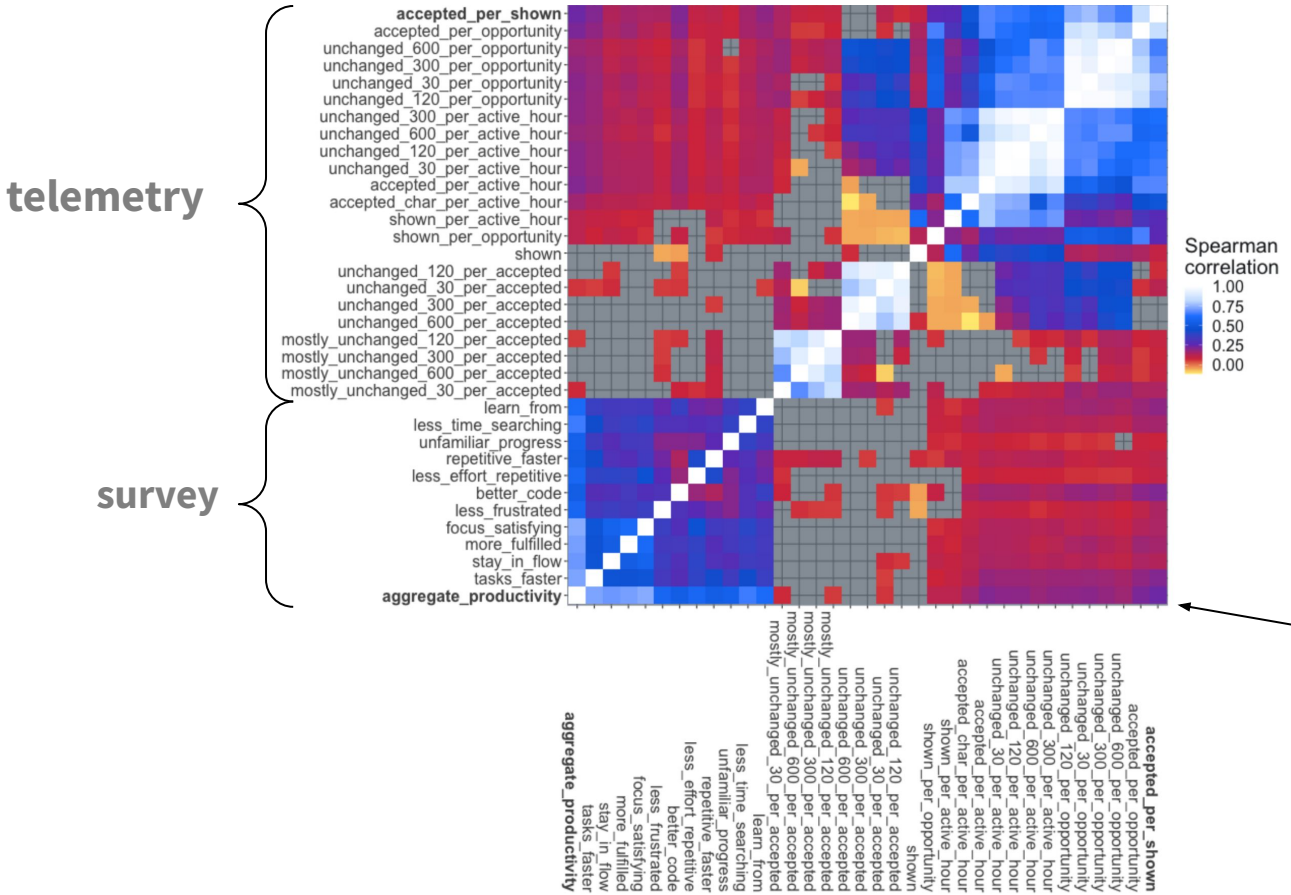
- A study run by GitHub about use of Copilot and its effect on productivity.
- Surveyed users of Copilot and matched to their Copilot telemetry data over a 4 week period, resulting in 2,000 users.



Productivity Assessment of Neural Code Completion (Ziegler '22)

- Productivity measures in the **survey**:
  - Self-reported productivity ("I am more productive")
  - Satisfaction, performance, communication, and efficiency (all self-reported)
- IDE actions from **telemetry**:
  - Completions shown
  - Completions accepted
  - How much completions were edited
  - How long completions persisted unchanged in the editor

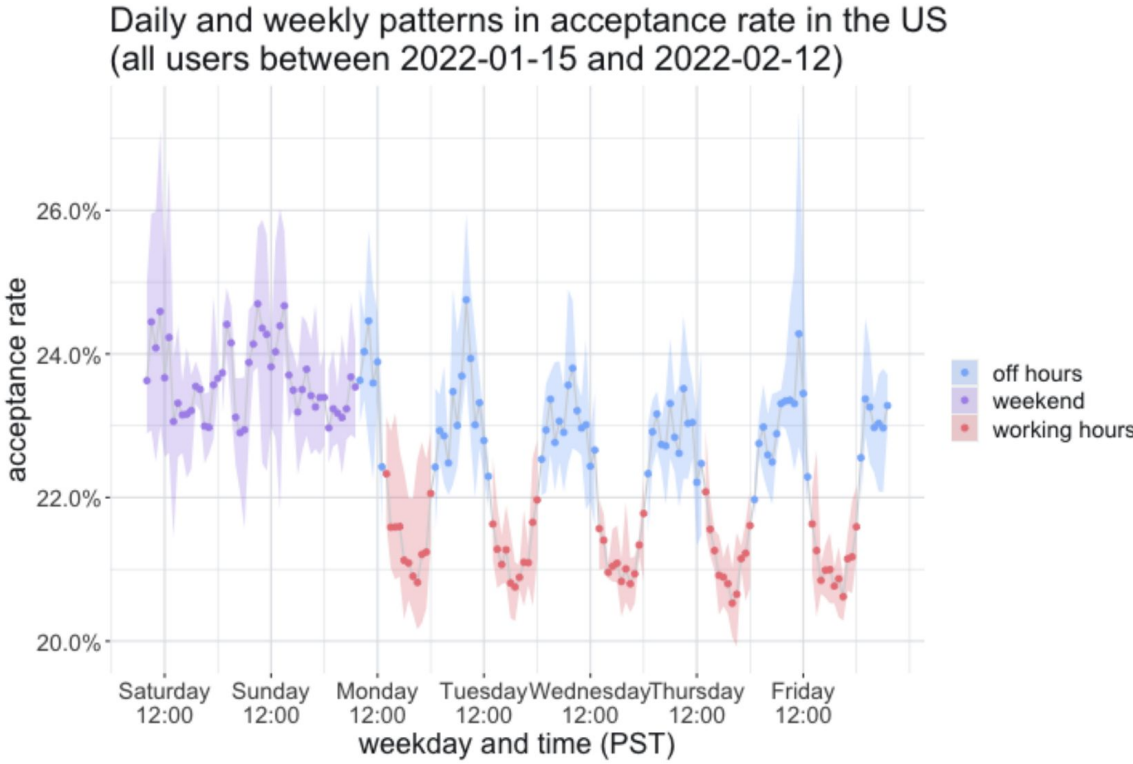




- Check correlation of all metrics: from telemetry and survey
- Telemetry events are correlated; survey are correlated
- Of telemetry events, accept rate is most correlated with perceived productivity in survey ($\varrho = 0.24$)

Productivity Assessment of Neural Code Completion (Ziegler '22)

- Acceptance rates are time-dependent



Experience Reports: Inputs and Outputs

- Writing prompts is hard
  - "*The comments used to prompt Copilot are just another very inefficient programming language*"
  - "*There is something valuable about being able to write at different levels of abstraction*"
- Copilot is trained on a mix of code ages and qualities
  - "*Copilot has made my code more verbose*"
  - "*Copilot provided code for using older versions of [a library]*"
  - *"Copilot showed me how to add structure in my code in unlikely places... I find myself discovering new API methods, either higher-level ones or ones that are*

to develop a sentiment classifier for Twitter posts matching certain keywords remarks, *"These kinds of things are possible not just because of co pilot [sic] but also because we have awesome libraries which have abstracted a lot of tough stuff."* This suggests that API design, not just for human developers but also as a target for large language models, will be important in the near and mid-term future.

Human Goals and System Capabilities

- Gulf of Execution
  - "How do I get the computer to do what I want"?
  - Long been a problem in HCI, e.g. text interfaces don't show visual affordances in the way that GUIs do (Brennan 1998)

- "Fuzzy abstraction problem"
  - Special case of the Gulf of Execution problem
  - “What can the system understand? What ‘syntax’ does it require?”
  - LLMs handle a wide range of abstractions, but noisily
    - Easy to start, hard to master

What About Cognitive Challenges?

In the context of programming activities, *exploratory programming*, where the goal is unknown or ill-defined (Kery & Myers, 2017; Sarkar, 2016), does not fit the framing of fuzzy abstraction matching (or indeed any of the variations of the gulf of execution problem). When the very notion of a crystallised user *intent* is questioned, or when the design objective is for the system to influence the intent of the user (as with much designerly and third-wave HCI work), the fundamental interaction questions change. One obvious role the system can play in these scenarios is to help users refine their own concepts (Kulesza et al., 2014) and decide what avenues to explore. Beyond noting that such activities exist, and fall outside the framework we have proposed here, we will not explore them in greater detail in this paper.

Experience Reports: Shifting Priorities

- Surface forms can be deceptive
  - *"[...] it has proved to be very good at producing superficially appealing output that can stand up not only to a quick scan, but to a moderately deep reading, but still falls apart on a more careful reading. [...] it's an uncanny valley type effect. [...] it's almost the most dangerous possible iteration of it, where it's good enough to fool a human functioning at anything other than the highest level of attentiveness but not good enough to be correct all the time. See also, the dangers of almost self-driving cars; either be self-driving or don't but don't expect halfway in between to work well."*
- Less time writing, more time debugging
  - *"I find I spend my time reviewing Copilot suggestions (which are mostly wrong) rather than thinking about code and actually doing the work."*
  - *"I turned off auto-suggest and that made a huge difference. Now I'll use it when I know I'm doing something repetitive that it'll get easily, or if I'm not 100% sure what I want to do and I'm curious what it suggests. This way I get the help without having it interrupt my thoughts with its suggestions."*
- Copilot is great at writing boilerplate code

Inadequate Metaphors

- As Search
  - Same: prompt-based, information asymmetry, several results, inexact solutions
  - Different:
    - Search has additional context: discussions, upvotes, images, …
    - Search has provenance: sources, dates, licenses,...
- As Compilation
  - E.g.: natural language “compiled to” source code
  - Same: "program" at a higher level of abstraction
  - Different:
    - With LLMs, we have to check the outputs!
    - With compilers, we don’t have to think about the lower-level (e.g. assembly).
- As Pair Programming
  - Driver (low-level, implementation) vs navigator (high-level, planning)
  - Same: AI as driver
  - Different:
    - With LLMs, you have to swap roles much more frequently (e.g., analyze driver’s outputs, write some code on your own). No social pressure.

Recap

- LLMs (e.g., via copilot) expand the scope and quality of code automation

- New *gulf of execution*: how to get the system to do what I want?
  - Difficult to prompt, introduces errors
  - Shifts time allocation while programming (e.g. more debugging)
  - Can be useful! (e.g., add structure, discover API methods, repetitive tasks)

- Past metaphors do not capture the full experience of “programming with AI”
  - Search
  - Compilation

    Pair programming

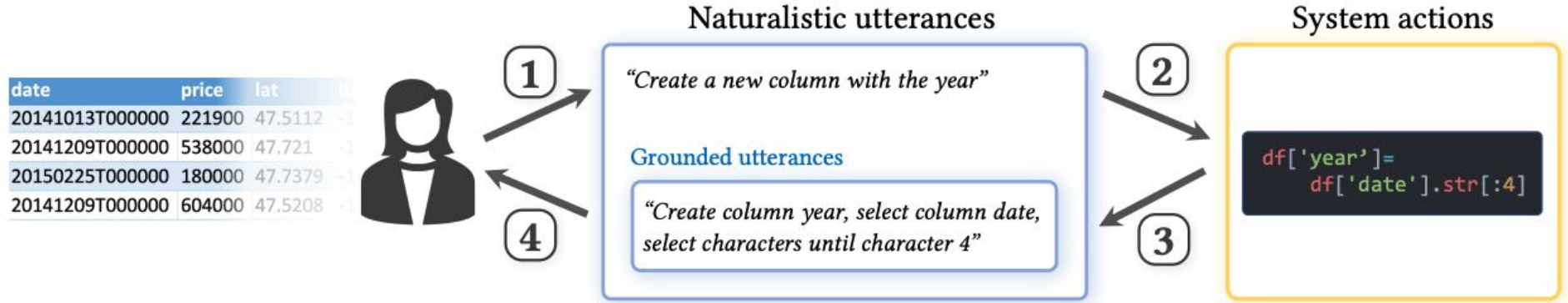
Open Issues

All of the above focused on professional programmers!
Issues for more novice users (based on Ragavan et al. 2022, a study on language-augmented spreadsheets):

1. Intent specification: help users think computationally or prompt unambiguously
2. Code correctness, quality, and (over)confidence: how to help novice users check, and trust, code?
3. Code comprehension and maintenance: generate code that's more contextual/user-tailored/lower cognitive load
4. Consequences of automation: what to do with saved time? How do ecosystems change? What about data feedback loops?
5. No code, and the dilemma of the direct answer: do users even want code, or just outputs?

Bridging the Abstraction Gap (Liu et al., CHI '23)

- Translate a user's utterance into a "grounded utterance" (fine-grained description) that matches the system-produced code more closely.
- The user sees the code's output, and can edit the grounded utterance to fix it if the output is wrong.
- Domain: making edits to tables via Pandas code.

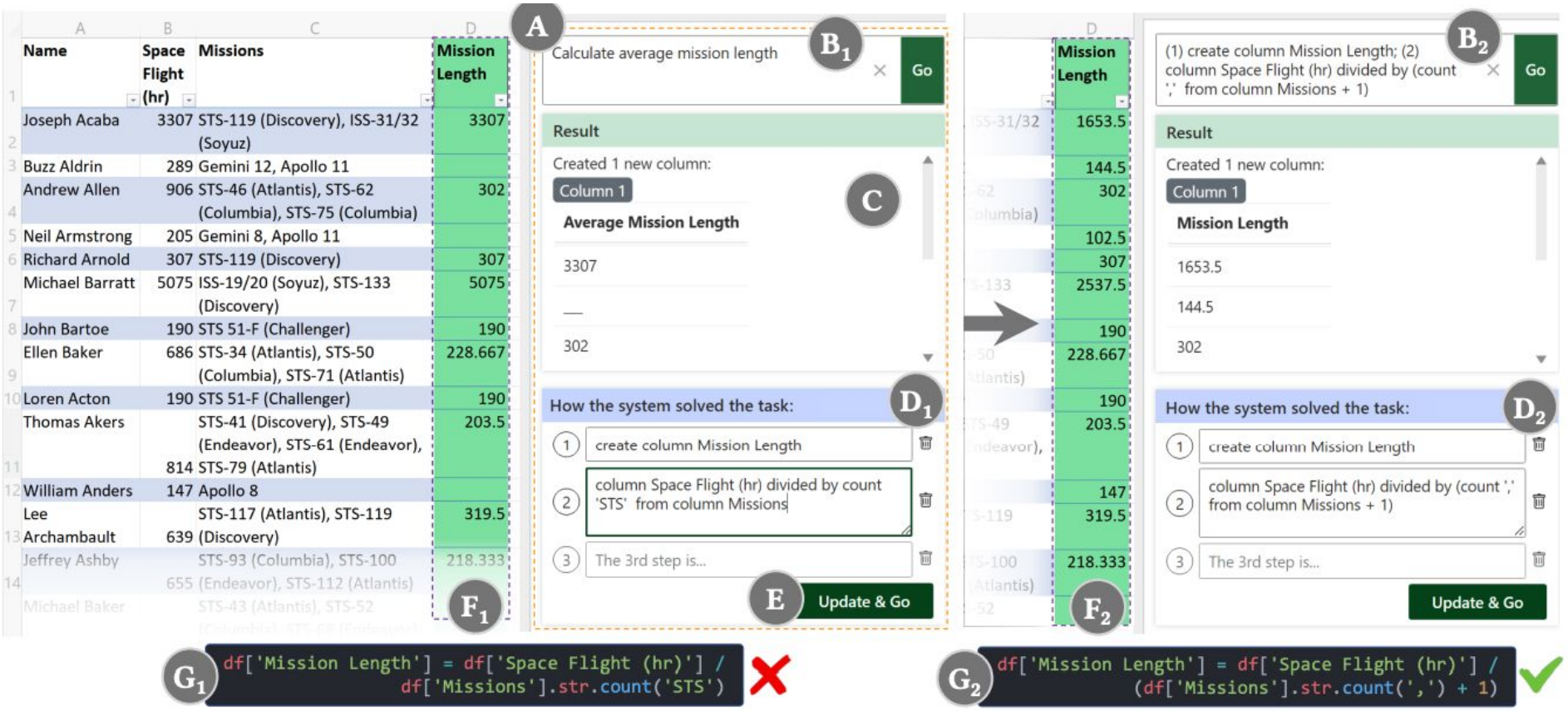


Bridging the Abstraction Gap (Liu et al., CHI '23)

In other words, our interface does not directly address *"this is*

tion. Our hypothesis is that exposure to such grounded examples leads to more effective and confident use of a natural language programming interface.

just asked it to do" (grounded abstraction matching). This differen-



	Name	Space Flight (hr)	Missions	Mission Length
2	Joseph Acaba	3307	STS-119 (Discovery), ISS-31/32 (Soyuz)	3307
3	Buzz Aldrin	289	Gemini 12, Apollo 11	
4	Andrew Allen	906	STS-46 (Atlantis), STS-62 (Columbia), STS-75 (Columbia)	302
5	Neil Armstrong	205	Gemini 8, Apollo 11	
6	Richard Arnold	307	STS-119 (Discovery)	307
7	Michael Barratt	5075	ISS-19/20 (Soyuz), STS-133 (Discovery)	5075
8	John Bartoe	190	STS 51-F (Challenger)	190
9	Ellen Baker	686	STS-34 (Atlantis), STS-50 (Columbia), STS-71 (Atlantis)	228.667
10	Loren Acton	190	STS 51-F (Challenger)	190
11	Thomas Akers	814	STS-41 (Discovery), STS-49 (Endeavor), STS-61 (Endeavor), STS-79 (Atlantis)	203.5
12	William Anders	147	Apollo 8	
13	Lee Archambault	639	STS-117 (Atlantis), STS-119 (Discovery)	319.5
14	Jeffrey Ashby Michael Baker	655	STS-93 (Columbia), STS-100 (Endeavor), STS-112 (Atlantis) STS-43 (Atlantis), STS-52	218.333

Bridging the Abstraction Gap (Liu et al., CHI '23)

- Showing users "grounded utterances" didn't affect their task completion rate or the time to complete the task.
- Anecdotally, grounded utterances increased some users' trust and confidence, helped shaped mental models

> In general, participants thought that the grounded utterances *"made it easy to check your work"* (P4), highly *"programmable"* (P5), and *"providing opportunities for you to modify and iterate over it"* (P26).

- Grounded utterances led people to adapt their language in different ways.

> Participants picked up vocabularies and styles of utterance from the grounded utterances, which would reliably get the system to work according to their intent (7/12). This was particularly helpful for non-programmers, for example, P13 recalled that through inter-

Grounded Copilot:
How Programmers Interact with Code-Generating Models

Sharddha Barke, Michael B. James, Nadia Polikarpova

OOPSLA (Object-oriented Programming, Systems, Languages, and Applications) '23

https://arxiv.org/abs/2206.15000

Grounded Theory Analysis

- Glaser and Strauss 1967. A bottom-up strategy for qualitative research
  - Exploratory; starts with a blank slate rather than existing hypotheses
  - Interleave analysis and data collection; use analysis to inform further experiments

- *Qualitative coding*: tag raw data (transcripts, videos, etc) with tags that explain it
  - *Open coding* (early stages): don't use a set of predefined codes, let researchers define as they go
  - *Axial coding:* aggregate and analyze codes to identify *conditions* and *strategies*
  - *Selective coding* (later stages): linking together codes and notes into a theory (I think)

Are there risks to having the analysis shape the experiments?

Experimental Setup

- Analyze interactions of 20 participants with Copilot
  - All were students or professionals in CS
  - About half had prior Copilot experience
- Process
  - 1 hour of training, then 20-40 minute core task
  - Not required to use Copilot, but encouraged
  - Talk through their interactions & have a semi-structured interview afterward

Task	Language(s)	Description	Purpose
Chat Server	Python/Rust	Implement core "business logic" of a chat application, involving a small state machine.	Investigate how Copilot aids in interpreting and implementing a human-language specification.
Chat Client	Python/Rust	Implement networking code for a chat application, using a custom cryptographic API and standard but often unfamiliar socket API.	Probe how Copilot can aid with a custom API.
Benford's Law	Rust & Python	Use Rust to generate two sequences—the Fibonacci sequence and reciprocals of sequential natural numbers; then plot these sequences using Python's `matplotlib`.	Collect data on a straightforward task (acceleration) and on an unfamiliar task (exploration).
Advent of Code	Python/Rust/ Haskell/Java	Implement a string manipulation task from a programming competition.	How will programmers test their Copilot-assisted code?

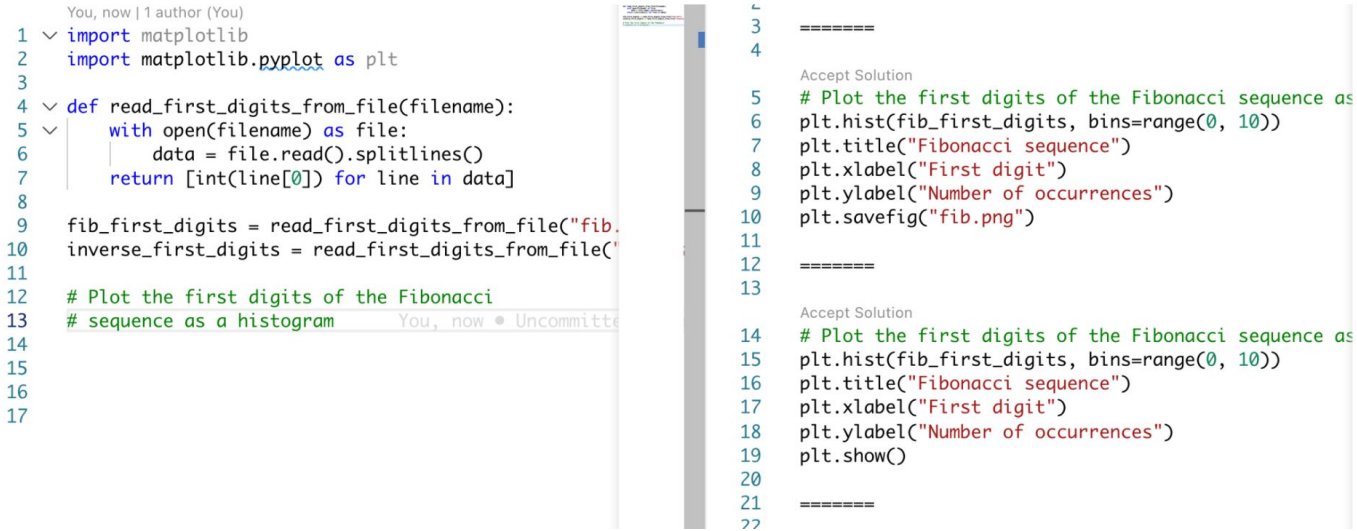
Two Mode Theory

- *Acceleration*: like autocomplete on steroids. Programmer knows what they want; Copilot types it faster

```
# 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(" => ")
```

Two Mode Theory

- *Exploration*: rely on Copilot to help plan actions, consider alternatives.



Acceleration Mode

- Programmer is driving. Use acceleration after decomposing the task.
  - Microtasks, e.g. *parse the input*, *compute the output*.
  - Can specify microtasks with e.g. type signatures and descriptive names
  - People can use acceleration even if they don't know the language or Copilot, ***as long as they know the algorithm***

- Copilot's suggestions should be short and focused.
  - Long suggestions break flow

- Validating suggestions: Pattern match for keywords; reject if they aren't present.

Exploration Mode: Overall

- Programmer lets Copilot drive, e.g. on novel tasks, or when they have less experience, or when code doesn't work.
  - Requires trust in the model, but can lead to automation bias:

> “I was trying to get Copilot to do it for me, maybe I should have given smaller tasks to Copilot and done the rest myself instead of depending entirely on Copilot.”

- Preference for comment prompts (more control than code prompts).
  - Comments are written especially for Copilot (and often deleted afterward!)
  - Evidence of adaptation to the model: rephrase comment if Copilot gets it wrong

Exploration Mode: Multiple Suggestions

- Willingness to spend time looking through solutions and combine them

> "I mostly just do a deep dive on the first one it shows me, and if that differs from my expectation, for example when it wasn't directly invoking the handshake function, I specifically look for other suggestions that are like the first one but do that other thing correctly."

- Substitutes for search engines, but need heuristics for trustworthiness

> "what would have been a StackOverflow search, Copilot pretty much gave that to me."

> "I'm pretty confident. I haven't used this socket library, but it seems Copilot has seen this pattern enough that, this is what I want."

- Cognitive overload & anchoring biases

> "It might be nice if it could highlight what it's doing or which parts are different, just something that gives me clues as to why I should pick one over the other."

Room for better interface design?

Exploration Mode: Validation Strategies

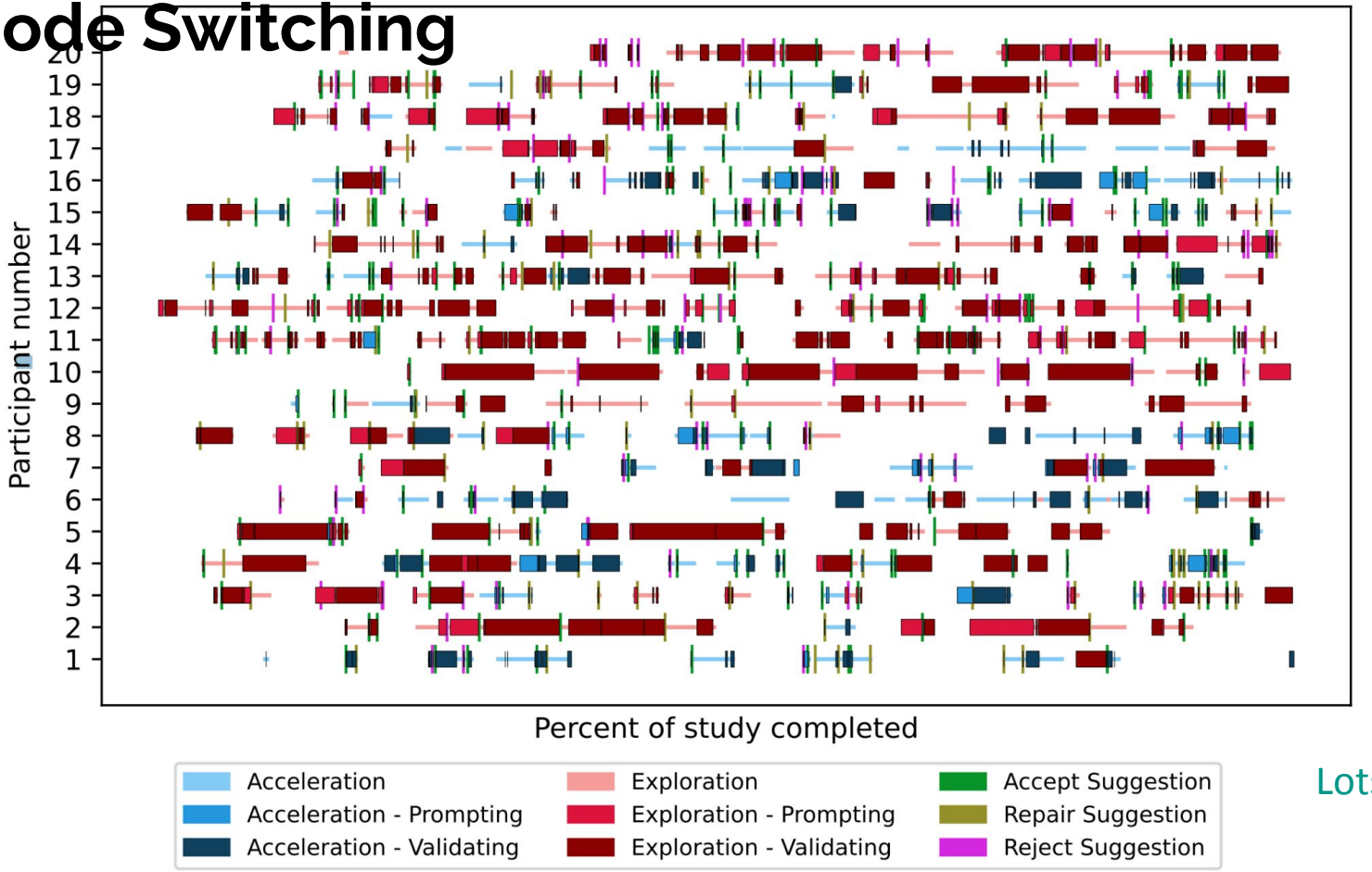
- Examination (most common), execution (REPL), static analysis (in Rust, etc), documentation (in-browser, or on the web)
- More willing to accept and edit code than in acceleration mode

> "If I'm in a mode where I want to rip apart a solution and use it as a template then I can look at the multi-suggestion pane and select whichever suits my needs."

- Cognitive load is an issue

> "I don't see the error immediately and unfortunately because this is generated, I don't understand it as well as I feel like I would've if I had written it. I find reading code that I didn't write to be a lot more difficult than reading code that I did write, so if there's any chance that Copilot is going to get it wrong, I'd rather just get it wrong myself because at least that way I understand what's going on much better."

Mode Switching



Lots of individual variation

Recommendations

- Better inputs
  - people don't understand what Codex can "see"
  - people want a dedicated syntax, i.e. force using a particular function or data structure
  - Write code in a language you're familiar with; have Codex translate it
- Better outputs
  - Awareness of the interaction mode: have the tool adjust to the user (e.g. shorter suggestions)
    - See Johnson et al. 2023, R-U-Sure for a tool for uncertainty-aware suggestion truncation
  - Tools for exploring multiple suggestions
    - Glassman et al., 2015 OverCode
  - Generate code with holes
    - See R-U-Sure again
  - Always-on validation (continual testing, or display variable values)

Discussion

- How can we use interaction data to train and improve models?
- Is code the best medium for suggestions?
- How might the sample size impact results?
- As copilot improves, how might the use of copilot (e.g. for acceleration and exploration) change?
- What kinds of personalization might be helpful? What research questions would pursuing these translate to?

Discussion

- Carolyn's comments
  - Toward a first-person view: cognitive task analyses
  - Other desiderata: Enhancing creativity, prompting thinking, supporting learning
- Some interesting looking papers (summaries from Austin Henley, pt 1, pt 2)
  - Studying the Effect of AI Code Generators on Supporting Novice Learners in Introductory Programming
  - “What It Wants Me To Say”: Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models
  - VizProg: Identifying Misunderstandings By Visualizing Students’ Coding Progress