

ai-assisted programming today and tomorrow

Nadia Polikarpova
PLDI'24



GitHub Copilot



Chat GPT



Amazon CodeWhisperer

...



GitHub Copilot

1.3M paid users
as of Feb 2024

generates 30-40% of the code
when in active use

today

how do programmers
use AI assistants?
what are the pain points?

tomorrow

how do we make AI
assistants more usable?

today

grounded copilot:
grounded theory
of AI-assisted programming

[\[Barke et al, OOPSLA'23\]](#)

other studies

productivity
usage patterns
code quality

tomorrow

leap:
validating AI-generated code
with live programming

[\[Ferdowsifard et al, CHI'24\]](#)

other designs

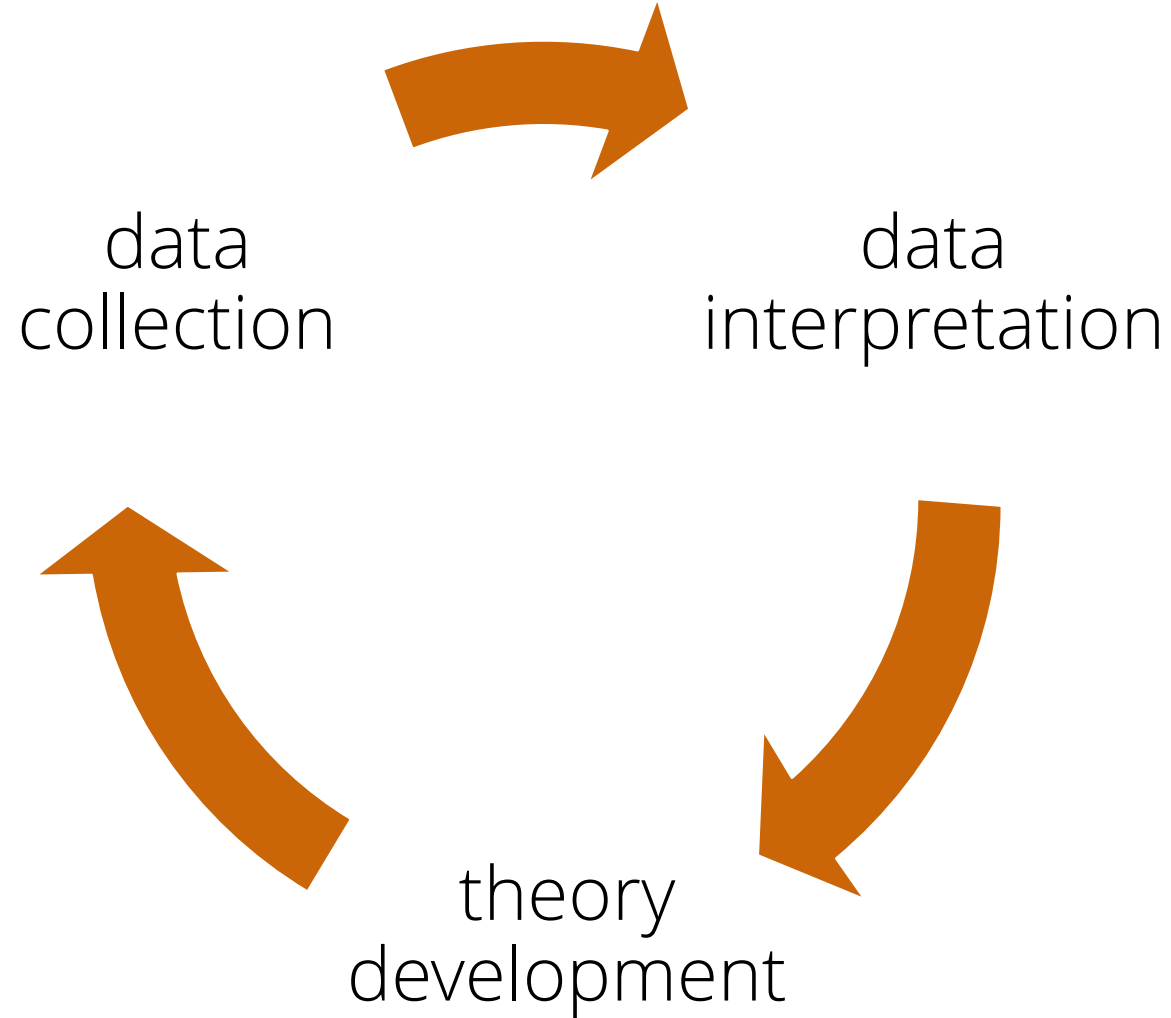
in-situ explanations
exploring suggestion spaces

today

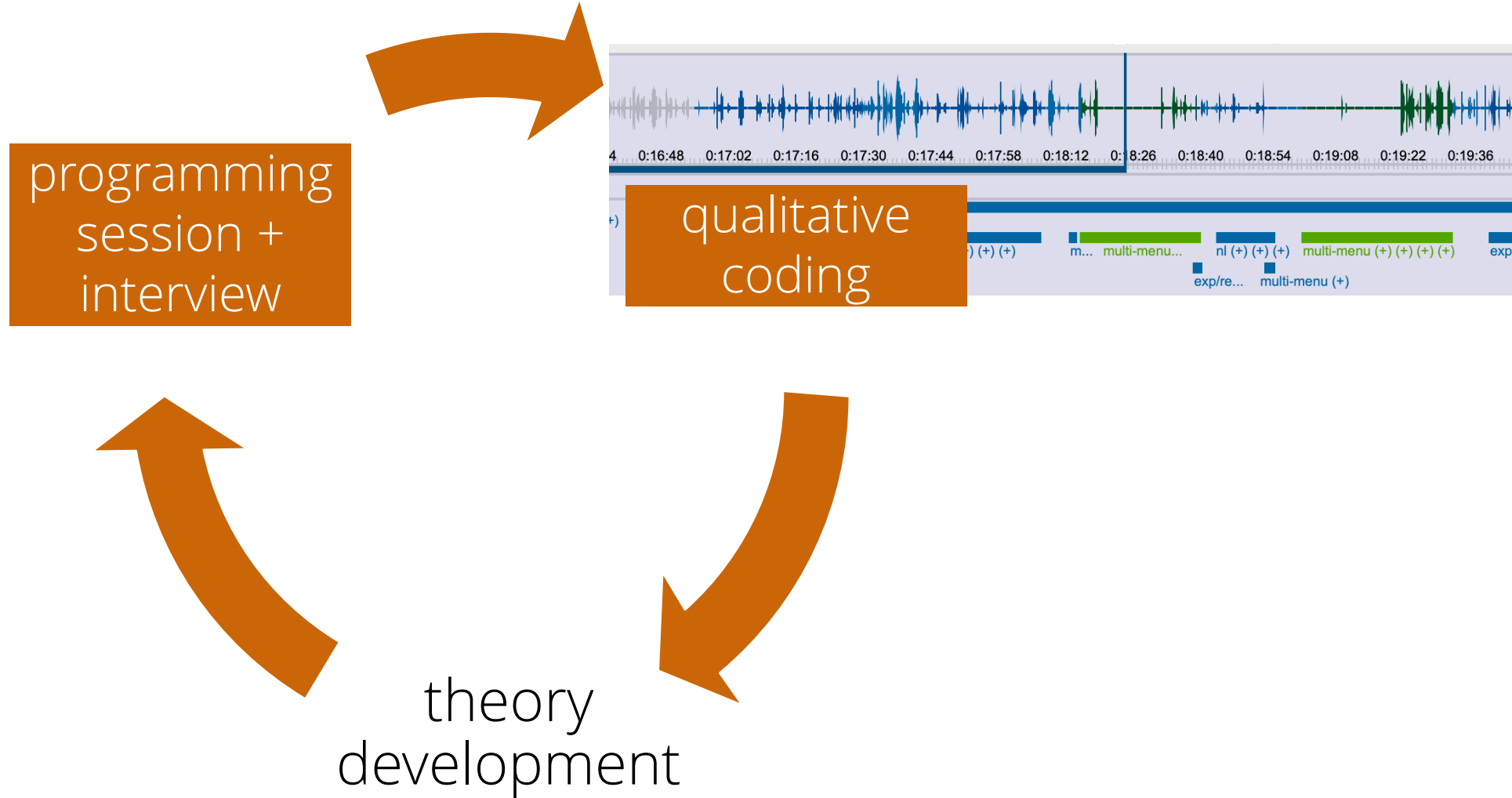
grounded copilot:
grounded theory
of AI-assisted programming

[\[Barke et al, OOPSLA'23\]](#)

grounded theory



grounded theory



tasks

chat server

business logic of a chat app

Python/Rust

chat client

networking + custom crypto API

Python/Rust

benford's law

familiar algorithm + matplotlib

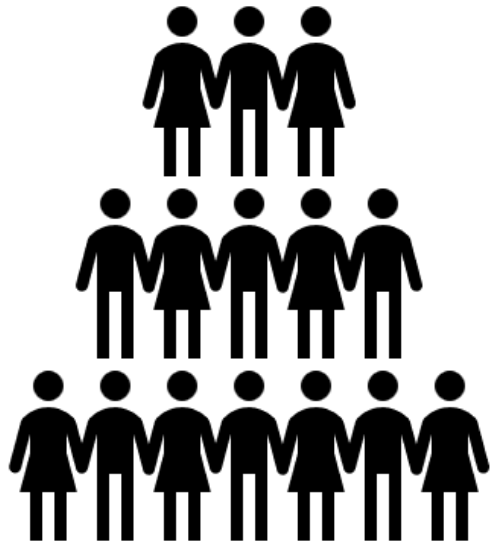
Rust + Python

string rewriting

competition task, easy to test

Python/Rust/Haskell/Java

participants



n = 20

occupation:

15 academia / 5 industry

language proficiency:

occasional / regular / professional

prior Copilot experience:

9 no / 11 yes

programming, fast and slow

acceleration

VS

exploration

autocomplete++

StackOverflow++

programmer has a plan
copilot helps them get there faster

programmer is lost
copilot suggests potential solutions

programming, fast and slow

acceleration

autocomplete++

programmer has a plan
copilot helps them get there faster

acceleration: example

programmer: broke down the task,
has a good idea for this function

```
# 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 = |
```

pauses
(unintentional prompting)

acceleration: example

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

programmer: "pattern-matches"
suggestion against expectations;
quickly accepts,
without leaving flow

copilot: auto-completes current logical unit (line of code)

programming, fast and slow

acceleration

VS

exploration

autocomplete++

StackOverflow++

programmer has a plan
copilot helps them get there faster

programmer is lost
copilot suggests potential solutions

exploration: example

programmer:
unfamiliar with matplotlib

```
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 • Uncommitt
14
15
16
17
```

intentionally prompts
with a comment;
invokes side panel

exploration: example

programmer: carefully examines suggestions;
compares to gauge confidence in API usage

```
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("fib.")
11
12 # Plot the first digits of the Fibonacci
13 # sequence as a histogram
14
15
16
17
```

copilot suggests multiple alternatives

```
4  =====
5  Accept Solution
6  # Plot the first digits of the Fibonacci sequence as
7  plt.hist(fib_first_digits, bins=range(0, 10))
8  plt.title("Fibonacci sequence")
9  plt.xlabel("First digit")
10 plt.ylabel("Number of occurrences")
11 plt.savefig("fib.png")
12
13 =====
14 Accept Solution
15 # Plot the first digits of the Fibonacci sequence as
16 plt.hist(fib_first_digits, bins=range(0, 10))
17 plt.title("Fibonacci sequence")
18 plt.xlabel("First digit")
19 plt.ylabel("Number of occurrences")
20 plt.show()
21
22 =====
23 Accept Solution
24 # Plot the first digits of the Fibonacci sequence as
25 plt.hist(fib_first_digits, bins=10, range=(0, 10))
26 plt.title("Fibonacci sequence")
27 plt.xlabel("First digit")
28 plt.ylabel("Number of occurrences")
29 plt.savefig("fib.png")
30
```

might cherry-pick parts
from different suggestions

validates code by executing
or consulting documentation

programming, fast and slow

acceleration

VS

exploration

autocomplete++

StackOverflow++

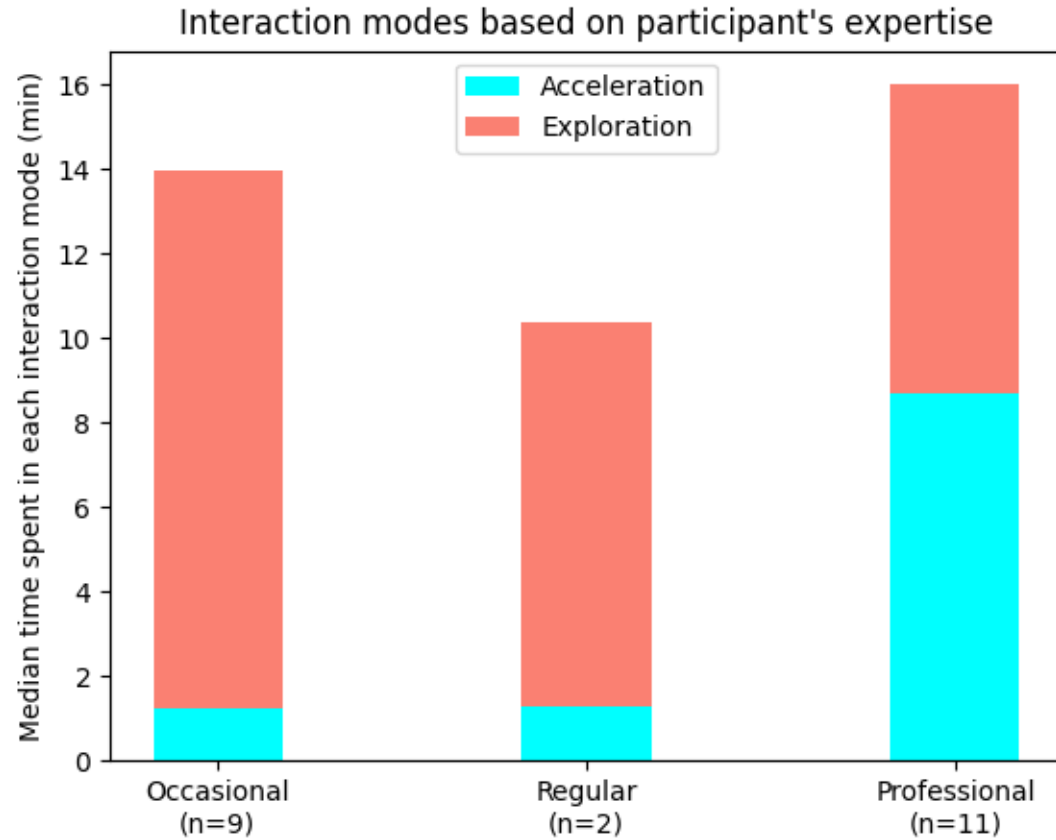
programmer has a plan
copilot helps them get there faster

programmer is lost
copilot suggests potential solutions

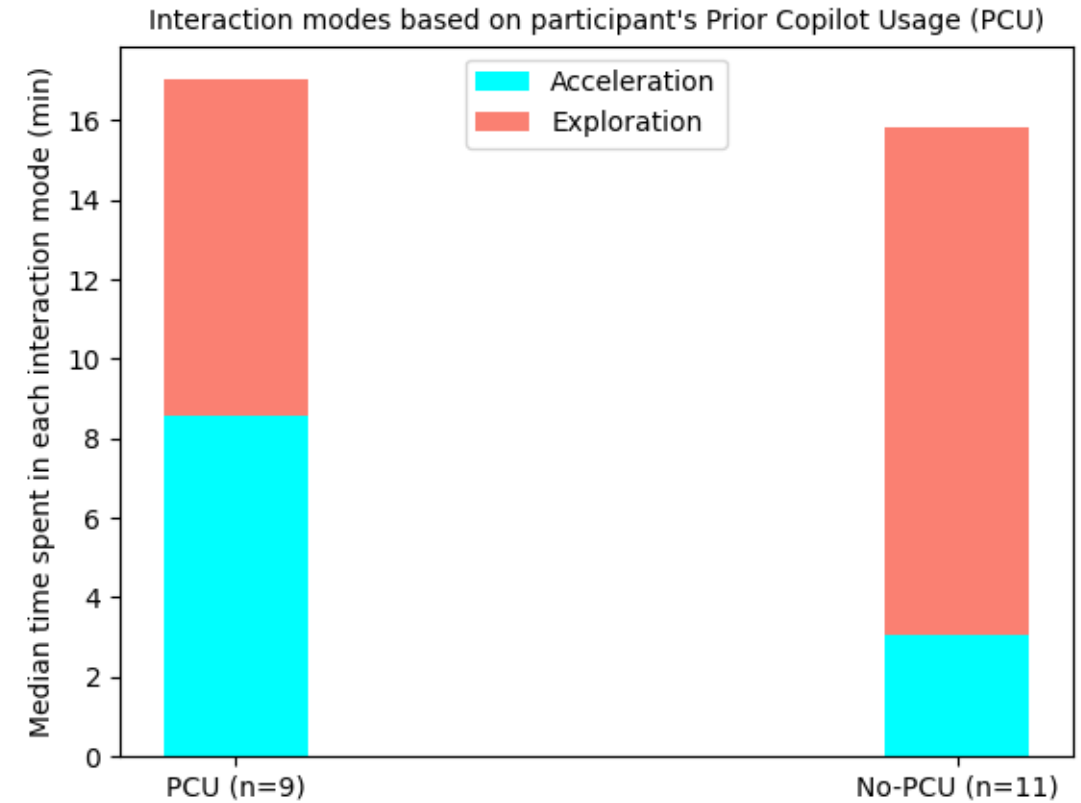
acceleration

vs

exploration



proficient language users accelerate more



new copilot users explore more

acceleration

unintentional

“pattern matching”

unit of focus
(sub-expression / statement)

unwilling to edit

vs

prompting

validation

scope

mismatch
tolerance

exploration

intentional with comments /
invoke side panel

explicit validation via
examination / execution /
documentation

entire function +
multiple alternatives

willing to edit / debug /
“rip apart” / cherry-pick

today

grounded copilot:
grounded theory
of AI-assisted programming

other studies
productivity
usage patterns
code quality

productivity

[\[Ziegler et al, MAPS'22\]](#)

analysis of 2531 survey responses + telemetry from Copilot
measure perceived productivity

results

programmers perceive themselves
as more productive

correlated with acceptance rate

average acceptance rate ~30%

productivity (objective)

[\[Vaithilingam et al, CHI EA'22\]](#)

24 participants (mostly students)
3 programming tasks (easy to hard)
Copilot vs IntelliSense

results

no improvement in task completion rate
or time
but most participants preferred Copilot

[\[Peng et al, arXiv'23\]](#)

95 developers recruited through UpWork
task: HTTP server in JavaScript
Copilot vs regular IDE

results

completion time improved by 55.8%
rate also improved but not significantly

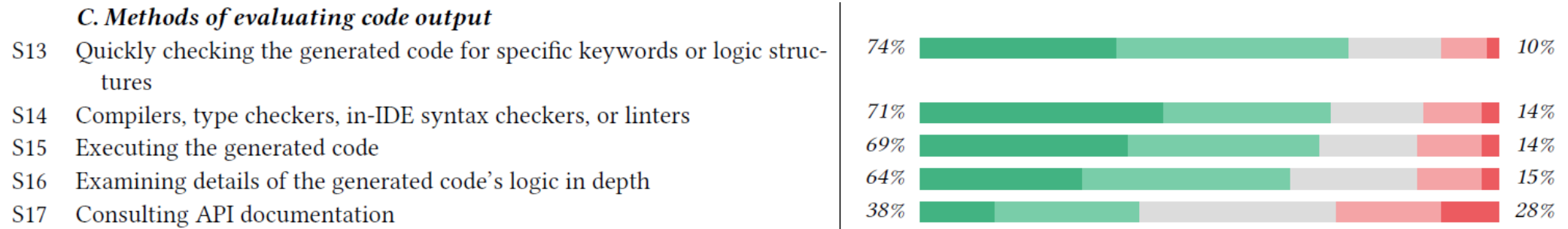
usage patterns

[\[Liang et al, ICSE'24\]](#)

survey of 410 developers using Copilot / ChatGPT / CodeWhisperer /etc
quantitative data to complement our findings

23% of developers have trouble validating suggestions

for example: prevalence of validation strategies related to their time cost



requests: more control (context, suggestion length), chat, auto-validation, explanations, multiple suggestions, personalization

usage patterns

[\[Mozannar et al, CHI'24\]](#)

observed 21 programmers using Copilot

results

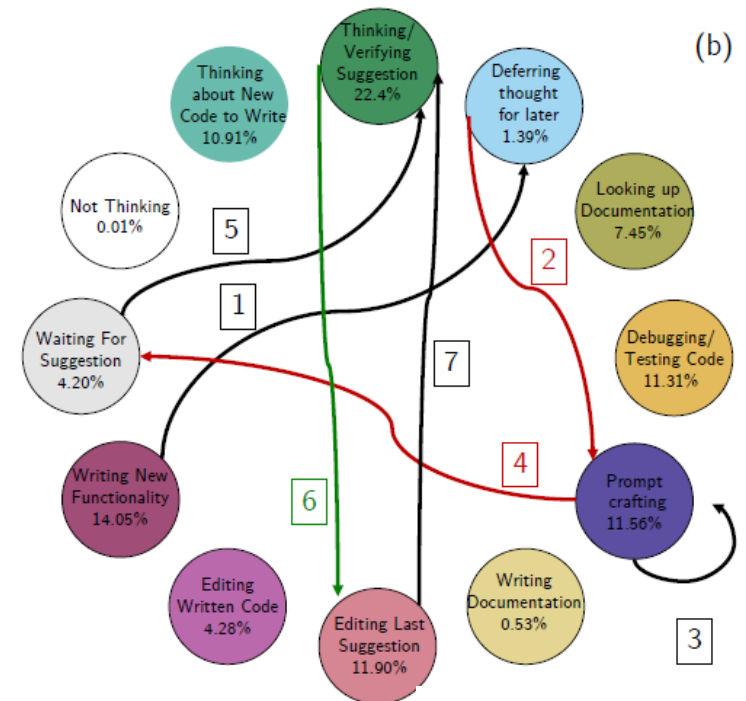
developed the CUPS taxonomy of user states
refinement of our two modes

collected stats on prevalence of states
and transitions

users spend the most time (22.4%)

validating suggestions

users often validate after “accepting”
(e.g. to see syntax highlighting)



code quality

[\[Kabir et al, CHI'24\]](#)

analysis of ChatGPT's answers to 517 StackOverflow questions
asked 12 programmers to rate the answers (ChatGPT vs Human)

results

52% of ChatGPT answers contain misinformation (25% incorrect code)

users preferred ChatGPT answers 35% of the time

77% of those answers were incorrect

users overlooked misinformation in the ChatGPT answers 39% of the time

requests: validation support, visualize uncertainty

code quality (security)

[\[Perry et al, CCS'23\]](#)

47 participants solve five security-related programming tasks in three languages (Python, JavaScript, and C) with and without AI

results

AI-assisted users produced less secure code

AI-assisted users who produced insecure code believed their code was secure and trusted the AI more



today

how do programmers
use AI assistants?
what are the pain points?

tomorrow

how do we make AI
assistants more usable?

programming = reviewing

"In the context of Copilot, there is a shift from writing code to understanding code"

Taking Flight with Copilot, ACM Queue, Dec 22

programmers spend **most of their time** validating generated code
context switching between writing and validating is **distracting**
programmers often **get it wrong** and **don't know it**

we must design for reviewing

lower the cost of validation

avoid distractions

develop appropriate trust

tomorrow

leap:

validating AI-generated code
with live programming

other designs

in-situ explanations
exploring suggestion spaces

leap

lowers the cost of validation by execution
using live programming

demo

```

2
3 def dominant_bigram(s):
4     '''
5     A Return the most common bigram in string s.
6     '''
7     ## ---
8     res = ''
9     bigrams = {}
10    for i in range(0, len(s) - 1):
11        bigram = s[i] + s[i + 1]
12        if bigram not in bigrams:
13            bigrams[bigram] = 0
14            bigrams[bigram] += 1
15    max_value = max(bigrams.values())
16    for bigram in bigrams:
17        if bigrams[bigram] == max_value:
18            res = bigram
19    ## ---
20
21    return res
22
23
24 dominant_bigram("agctagta")

```

```

• # bigram
0 'ag'
1 'gc'
2 'ct'
3 'ta'
4 'ag'
5 'gc'
6 'ta'

```

```

• # bigrams
0 {'ag': 0}
1 {'ag': 1, 'gc': 0}
2 {'ag': 1, 'gc': 1, 'ct': 0}
3 {'ag': 1, 'gc': 1, 'ct': 1, 'ta': 0}
4
5 {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 1, 'gt': 0}
6

```

```

• # bigrams
0 {'ag': 1}
1 {'ag': 1, 'gc': 1}
2 {'ag': 1, 'gc': 1, 'ct': 1}
3 {'ag': 1, 'gc': 1, 'ct': 1, 'ta': 1}
4 {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 1}
5 {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 1, 'gt': 1}
6 {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 2, 'gt': 1}

```

```

• max_value
2

```

```

• # res
0 'ag'
1
2
3 'ta'
4

```

```

• rv
'ta'

```

Suggestion 2

B

Preview

```

res = {}
s = s.lower()
for i in range(0, len(s)):
    if s[i] in res:
        res[s[i]] += 1
    else:
        res[s[i]] = 1
print(res)
for i in res.keys():
    if res[i] < 2:
        res.pop(i)
print(res)

```

Suggestion 3

Preview

```

res = ''
bigrams = {}
for i in range(0, len(s) - 1):
    bigram = s[i] + s[i + 1]
    if bigram not in bigrams:
        bigrams[bigram] = 0
    bigrams[bigram] += 1
max_value = max(bigrams.values())
for bigram in bigrams:
    if bigrams[bigram] == max_value:

```

user study

no-LP

AI suggestions
+
terminal

LP

AI suggestions
+
live programming

research questions

how does **live programming** affect...

1. over- / under-reliance on AI
2. validation strategies
3. cognitive load

tasks

API-heavy

pandas

clean dataframe and compute stats
using pandas

multiple correct suggestions

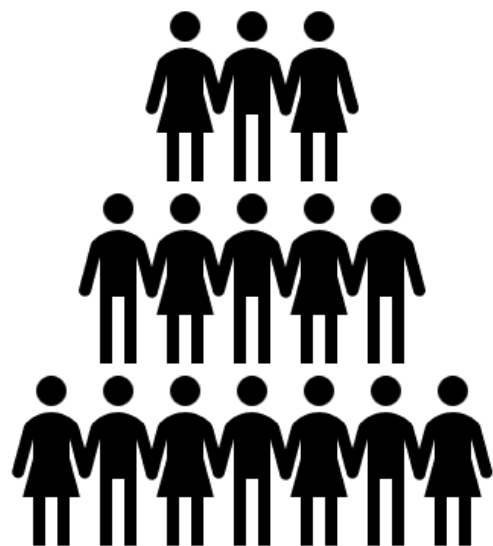
algorithmic

bigrams

find most frequent bigram in a string

no fully correct suggestions

participants



n = 17

occupation:

15 academia / 2 industry

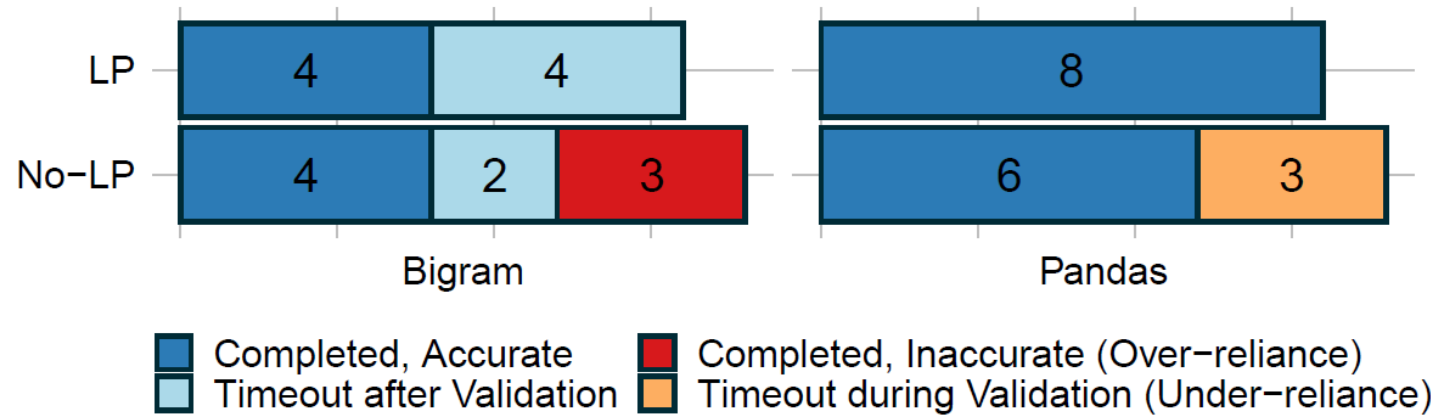
Python usage:

2 occasionally /

8 regularly /

7 almost every day

rq1: over-/under-reliance



6 no-LP vs 0 LP participants **mid-judged** correctness of their solution

by lowering the cost of validation,
leap reduces over-/under-reliance on AI

rq1: over-/under-reliance

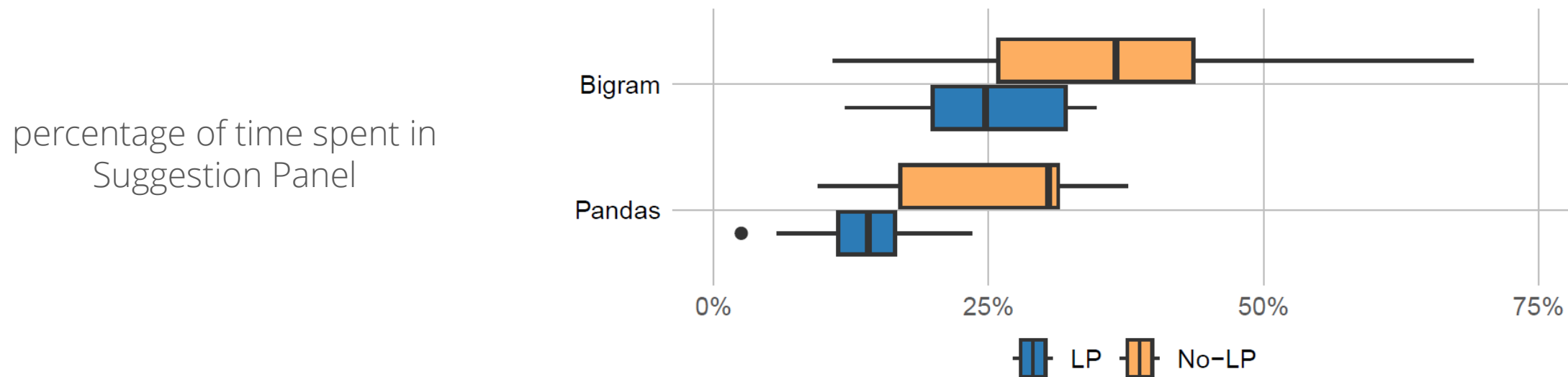
“it was **easy to understand** the behavior of a code suggestion because the little boxes on the side allowed for you to preview the results.” (P3)

“it **saved me the effort** of writing multiple print statements.” (P1)

6 no-LP vs 0 LP participants **mid-judged** correctness of their solution

by lowering the cost of validation,
leap reduces over-/under-reliance on AI

rq2: validation strategies

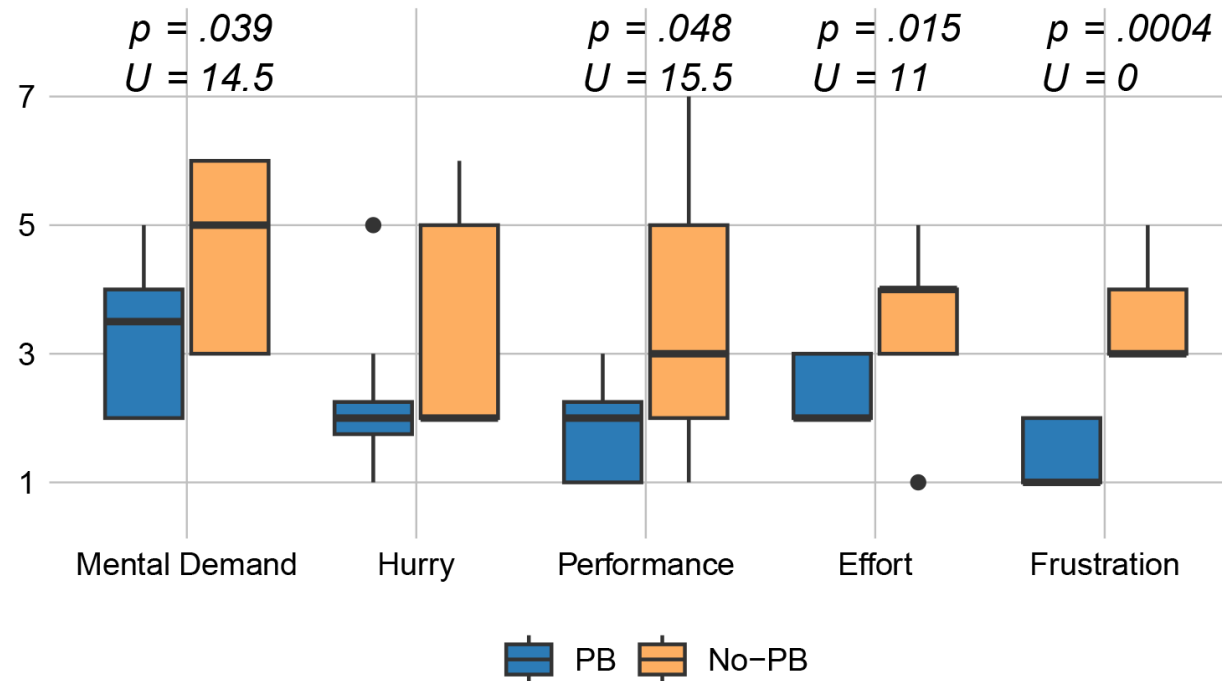


“I didn’t look too closely in the actual code,
I was *just looking at the runtime values* on the side.” (P1)

leap participants spent less time reading code

rq3: cognitive load

NASA TLX cognitive load metrics
on Pandas



leap significantly reduced cognitive load of AI-assisted programming on tasks amenable to validation by execution

tomorrow

leap:

validating AI-generated code
with live programming

other designs

in-situ explanations
exploring suggestion spaces

ivie: in-situ explanations

[\[Yan et al, CHI'24\]](#)

lightweight explanations appear automatically for generated code

```
df_all.merge(df_Apr, on='City', how='left', suffixes=('_all', '_apr'))
```

Combines
df_all with
another
DataFrame
df_Apr.

Only rows with
matching 'City'
values in both
DataFrames will
be merged.

Retains all rows
from df_all,
with matching
rows from
df_Apr. If no
match is found,
columns will be
NaN.

In case of
column name
conflicts, df_all
columns end
with _all and
df_Apr with
_apr.

ivie: in-situ explanations

[\[Yan et al, CHI'24\]](#)

lightweight explanations appear automatically for generated code

```
def visualize_data(df, max_temp_city, max_rain_city):  
    fig, ax = plt.subplots(2, 1, figsize=(14, 10))  
  
    df[df['City'] == max_temp_city]['Temperature'].plot(ax=ax[0])  
    ax[0].set_title(f'Yearly Average Temperature for {max_temp_city}')  
    ax[0].set_xlabel('Year')  
    ax[0].set_ylabel('Temperature (°C)')  
    ax[0].yaxis.set_major_formatter(ticker.FormatStrFormatter('%0.1f'))  
  
    df[df['City'] == max_rain_city]['Rainfall'].plot(ax=ax[1], color='green')  
    ax[1].set_title(f'Yearly Average Rainfall for {max_rain_city}')  
    ax[1].set_xlabel('Year')  
    ax[1].set_ylabel('Rainfall (mm)')  
    ax[1].yaxis.set_major_locator(ticker.MaxNLocator(nbins=6, integer=True))  
  
    plt.tight_layout()  
    plt.show()
```

1. Creates a figure with two vertical subplots.

2. Plot the temperature data for the city with the highest temperature in the top subplot.

3. Plot the rainfall data for the city with the highest rainfall in the bottom subplot.

4. Format the figure and display it.

ivie: in-situ explanations

[\[Yan et al, CHI'24\]](#)

lightweight explanations appear automatically for generated code

results: study participants

answered comprehension questions **more correctly** with Ivie

experienced **less mental load** and reported being **less distracted** and overwhelmed than with baseline (chat)

exploring suggestion spaces

why show multiple suggestions?

more likely to find what you want

absence of desired solution is the main reason

programmers don't use AI assistants [\[Liang et al, ICSE'24\]](#)

reduce over-reliance by exposing choices

variation theory: we learn better when exposed

to multiple instances of a concept with small differences

see AI-resilient interfaces [\[Glassman et al\]](#)

exploring suggestion spaces

but... existing interfaces are terrible

hard to get a high-level overview

hard to see what is common/different
between suggestions

```
1
2
3 =====
4
5 Accept Solution
6 # Plot the first digits of the Fibonacci sequence as
7 plt.hist(fib_first_digits, bins=range(0, 10))
8 plt.title("Fibonacci sequence")
9 plt.xlabel("First digit")
10 plt.ylabel("Number of occurrences")
11 plt.savefig("fib.png")
12
13 =====
14
15 Accept Solution
16 # Plot the first digits of the Fibonacci sequence as
17 plt.hist(fib_first_digits, bins=range(0, 10))
18 plt.title("Fibonacci sequence")
19 plt.xlabel("First digit")
20 plt.ylabel("Number of occurrences")
21 plt.show()
22
23 =====
24
25 Accept Solution
26 # Plot the first digits of the Fibonacci sequence as
27 plt.hist(fib_first_digits, bins=10, range=(0, 10))
28 plt.title("Fibonacci sequence")
29 plt.xlabel("First digit")
30 plt.ylabel("Number of occurrences")
31 plt.savefig("fib.png")
32
```

exploring suggestion spaces

[\[Gero et al, CHI'24\]](#)*

*for text, not code

```
res = None
d = {}
for i in range(len(s)-1):
    if s[i]+s[i+1] in d:
        d[s[i]+s[i+1]] += 1
    else:
        d[s[i]+s[i+1]] = 1
if d:
    res = max(d, key=d.get)
```

```
res = ''
bigrams = {}
for i in range(0, len(s) - 1):
    bigram = s[i] + s[i + 1]
    if bigram not in bigrams:
        bigrams[bigram] = 0
    bigrams[bigram] += 1
max_value = max(bigrams.values())
for bigram in bigrams:
    if bigrams[bigram] == max_value:
        res = bigram
```

```
bigram_dict = {}
for i in range(0, len(s) - 1):
    bigram = s[i] + s[i + 1]
    if bigram in bigram_dict:
        bigram_dict[bigram] += 1
    else:
        bigram_dict[bigram] = 1
max_count = 0
res = ''
for key, value in bigram_dict.items():
    if value > max_count:
        max_count = value
        res = key
```

```
s = s.lower().replace(' ', '')
res = ''
max = 0
for i in range(len(s) - 1):
    if s.count(s[i] + s[i + 1]) > max:
        max = s.count(s[i] + s[i + 1])
        res = s[i] + s[i + 1]
```

```
res = {}
s = s.lower()
for i in range(0, len(s)):
    if s[i] in res:
        res[s[i]] += 1
    else:
        res[s[i]] = 1
print(res)
for i in res.keys():
```

exploring suggestion spaces

[\[Gero et al, CHI'24\]](#)*

+ Ivie?

```
res = ''
bigrams = {}
for i in range(0, len(s) - 1):
    bigram = s[i] + s[i + 1]
    if bigram not in bigrams:
        bigrams[bigram] = 0
    bigrams[bigram] += 1
max_value = max(bigrams.values())
for bigram in bigrams:
    if bigrams[bigram] == max_value:
        res = bigram
```

```
bigram_dict = {}
for i in range(0, len(s) - 1):
    bigram = s[i] + s[i + 1]
    if bigram in bigram_dict:
        bigram_dict[bigram] += 1
    else:
        bigram_dict[bigram] = 1
max_count = 0
res = ''
for key, value in bigram_dict.items():
    if value > max_count:
        max_count = value
        res = key
```

initializes a dictionary

stores bigram counts in the dictionary

finds the highest count and stores corresponding bigram in **res**

```
''
```

```
1]) > max:
+ s[i + 1])
1]
```

```
res = {}
s = s.lower()
for i in range(0, len(s)):
    if s[i] in res:
        res[s[i]] += 1
    else:
        res[s[i]] = 1
print(res)
for i in res.keys():
```

questions for pl folks

ai assistants: curse or blessing for new languages?

can we use pl techniques to help with validation /
comprehension?

can we design languages to be ai-friendly? ai-resilient?

today

how do programmers
use AI assistants?
what are the pain points?

tomorrow

how do we make AI
assistants more usable?

thanks to students & collaborators!



Michael James



Shraddha Barke



Kasra Ferdowsi



Lisa Huang



Emmanuel Anaya
Gonzalez



Sorin Lerner



Elena Glassman