ai-assisted programming today and tomorrow

Nadia Polikarpova
PLDI’24
1.3M paid users as of Feb 2024

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

Github Copilot
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

data collection

data interpretation

theory development
grounded theory

programming session + interview

qualitative coding

theory development
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

occupation: 15 academia / 5 industry

language proficiency: occasional / regular / professional

prior Copilot experience: 9 no / 11 yes

n = 20
programming, fast and slow

acceleration  VS  exploration

autocomplete++

programmer has a plan
copilot helps them get there faster

StackOverflow++

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

```python
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
  autocomplete++
  programmer has a plan
  copilot helps them get there faster

VS

exploration
  StackOverflow++
  programmer is lost
  copilot suggests potential solutions
exploration: example

programmer:
unfamiliar with matplotlib

```
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
```
exploration: example

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

copilot suggests multiple alternatives

might cherry-pick parts from different suggestions

validates code by executing or consulting documentation
programming, fast and slow

**acceleration**
- autocomplete++
  - programmer has a plan
  - copilot helps them get there faster

**VS**

**exploration**
- StackOverflow++
  - programmer is lost
  - copilot suggests potential solutions
acceleration vs exploration

proficient language users accelerate more

new copilot users explore more
<table>
<thead>
<tr>
<th>acceleration</th>
<th>vs</th>
<th>exploration</th>
</tr>
</thead>
<tbody>
<tr>
<td>unintentional</td>
<td>prompting</td>
<td>intentional with comments / invoke side panel</td>
</tr>
<tr>
<td>“pattern matching”</td>
<td>validation</td>
<td>explicit validation via examination / execution / documentation</td>
</tr>
<tr>
<td>unit of focus</td>
<td>scope</td>
<td>entire function + multiple alternatives</td>
</tr>
<tr>
<td>(sub-expression / statement)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unwilling to edit</td>
<td>mismatch tolerance</td>
<td>willing to edit / debug / “rip apart” / cherry-pick</td>
</tr>
</tbody>
</table>
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

A survey of 410 developers using Copilot, ChatGPT, CodeWhisperer, etc., quantitatively complemented our findings. 23% of developers struggle to validate 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

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)

[Mozannar et al, CHI’24]
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
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.

---

References:

[Perry et al, CCS’23]
today

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

tomorrow

how do we make AI assistants more usable?
"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**
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
```
def dominant_bigram(s):
    '''
    Return the most common bigram in string s.
    '''

    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
    return res

dominant_bigram("agctagta")
```
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

algorithmic

bigrams

- find most frequent bigram in a string

- multiple correct suggestions

- no fully correct suggestions
participants

occupation:
15 academia / 2 industry

Python usage:
2 occasionally /
8 regularly /
7 almost every day

n = 17
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

percentage of time spent in Suggestion Panel

“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

lightweight explanations appear automatically for generated code

\texttt{df\_all.merge(df\_Apr, on='City', how='left', suffixes=('_all', '_apr'))}

- Combines \texttt{df\_all} with another DataFrame \texttt{df\_Apr}.
- Only rows with matching 'City' values in both DataFrames will be merged.
- Retains all rows from \texttt{df\_all}, with matching rows from \texttt{df\_Apr}. If no match is found, columns will be NaN.
- In case of column name conflicts, \texttt{df\_all} columns end with \texttt{\_all} and \texttt{df\_Apr} with \texttt{\_apr}.
ivie: in-situ explanations

[ Yan et al, CHI’24 ]

lightweight explanations appear automatically for generated code

```python
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('%.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

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
exploring suggestion spaces

```python
res = None
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)
```

```python
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
```

```python
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
```

```python
s = s.lower().replace(' ', '')
res = ''
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]
```

```python
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():
```

```python
res = {}
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
```

[Gero et al, CHI’24] *

*for text, not code
exploring suggestion spaces

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

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] > max:
        max = res[i]

[Gero et al, CHI'24]*
+ Ivie?
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