

Measuring The Runtime Performance Of C++ Code Written By Humans Using GitHub Copilot



Daniel
Erhabor



Sreeharsha
Udayashankar



Meiyappan
Nagappan

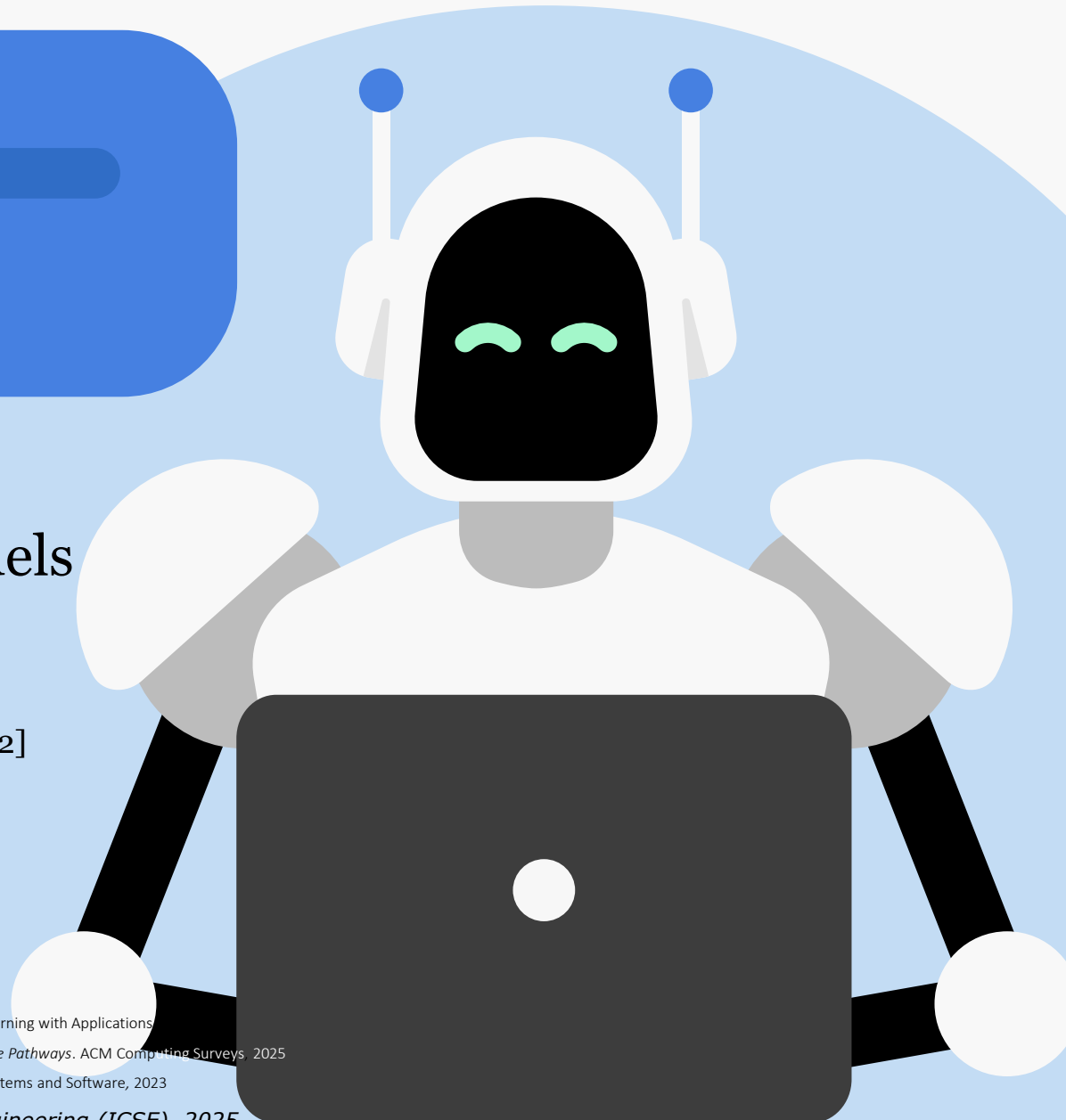


Samer
Al-Kiswany



UNIVERSITY OF
WATERLOO

Introduction



- Large Language Models
 - Chatbots [1]
 - Autonomous Agents [2]
 - Coding Assistants [3]

[1] E. Adamopoulou et al. *Chatbots: History, Technology, and Applications*, Machine Learning with Applications

[2] Z. Deng et al. *AI Agents Under Threat: A Survey of Key Security Challenges and Future Pathways*. ACM Computing Surveys, 2025

[3] A. Dakhel et al. *GitHub Copilot AI pair programmer: Asset or Liability?*, Journal of Systems and Software, 2023

GitHub CoPilot

- Coding Assistant
 - Powered by OpenAI Codex (GPT-3)*
 - Visual Studio extension



```
JS test.js 1 ●  
  
JS test.js > calculateDaysBetweenDates  
1 function calculateDaysBetweenDates(begin, end) {  
    var beginDate = new Date(begin);  
    var endDate = new Date(end);  
    var days = Math.round((endDate - beginDate) / (1000 * 60 * 60 * 24));  
    return days;  
}  
2
```

Copilot responding...

Ask Copilot

Q W E R T Y U I O P

* Now powered by choice of model between GPT-4o / Claude 3.5

GitHub CoPilot – Previous Studies

Usability / Functionality

Security

Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models

Priyan Vaithilingam
pvaithilingam@g.harvard.edu
Harvard University
USA

Tianyi Zhang
tiany@purdue.edu
Purdue University
USA

Elena L. Glassman
glassman@seas.harvard.edu
Harvard University
USA

ABSTRACT

Recent advances in Large Language Models (LLM) have made automatic code generation possible for real-world programming tasks in general-purpose programming languages such as Python. However, there are few human studies on the usability of these tools and how they fit the programming workflow. In this work, we conducted a within-subjects user study with 24 participants to understand how programmers use and perceive Copilot, a LLM-based code generation tool. We found that, while Copilot did not necessarily improve the task completion time or success rate, most participants preferred to use Copilot in daily programming tasks, since Copilot often provided a useful starting point and saved the effort of searching online. However, participants did face difficulties in understanding, editing, and debugging code snippets generated

A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges

Jenny T. Liang
Carnegie Mellon University
Pittsburgh, PA, USA
jliang@cs.cmu.edu

Chenyang Yang
Carnegie Mellon University
Pittsburgh, PA, USA
cyang3@cs.cmu.edu

Brad A. Myers
Carnegie Mellon University
Pittsburgh, PA, USA
bam@cs.cmu.edu

ABSTRACT

The software engineering community recently has witnessed widespread deployment of AI programming assistants, such as GitHub Copilot. However, in practice, developers do not accept AI programming

assistants. This leaves these tools, libraries, and the survey uses from active and motivated developers and recall storm poses why is do not unctio nolling the plications s, such as e tools to

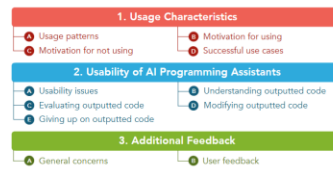


Figure 1: An overview of the topics covered in our usability study of AI programming assistants.

1 INTRODUCTION

The recent widespread deployment of AI programming assistants, such as GitHub Copilot [6] and ChatGPT [1], has introduced a new paradigm to building software that has taken the software engineering community by storm. Some current publications report that AI programming assistants are powerful enough to produce high-quality code suggestions for developers [59, 61]. While some

On Programming Variability with Large Language Model-based Assistant

Mathieu Acher
University of Rennes, INSA, IRISA,
Inria, IUF
Rennes, France
mathieu.acher@irisa.fr

José Galindo Duarte
University of Sevilla
Sevilla, Spain
jagalindo@us.es

Jean-Marc Jézéquel
University of Rennes, IRISA, Inria, IUF
Rennes, France
jean-marc.jezequel@irisa.fr

ABSTRACT

Programming variability is central to the design and implementation of software systems that can adapt to a variety of contexts and requirements, providing increased flexibility and customization. Managing the complexity that arises from having multiple features, variations, and possible configurations is known to be highly challenging for software developers. In this paper, we explore how large language model (LLM)-based assistants can support the programming of variability. We report on new approaches made possible with LLM-based assistants, like: features and variations can be implemented as prompts; augmentation of variability out of LLM-based domain knowledge; seamless implementation of variability in different kinds of artefacts, programming languages, and frameworks, at different binding times (compile-time or run-time). We are sharing our data (prompts, sessions, generated code, etc.) to support the assessment of the effectiveness and robustness of LLMs for variability-related tasks.

KEYWORDS

variability, programming, software product lines, generative AI, large language model

since developers should program, maintain, and test multiple features, code variations, and an exponential number of possible variants [4, 26, 32, 37, 47]. During the last decades, numerous languages, paradigms, and technologies have been developed to support systematic transformation of problem-level abstractions to software implementations. From the early days of generative programming [16] and software product line (SPL) engineering [4, 37, 47], the goal has been to automatically generate variants from a specification written in one or more textual or graphical domain-specific languages.

In this short and exploratory paper, we defend the idea that large language models (LLMs) can be leveraged to support the programming of variability and realize the early ambition of generative programming and SPL engineering. As experimented and reported in this paper, an emerging pattern is that LLMs act as a new variability compiler capable of transforming a high-level specification (prompt) into variable code, features, generators, configurable systems, or SPLs written in a given technological space.

LLMs are gaining momentum and are capable of tackling more and more problems from linguistics, maths, commonsense reasoning, biology, physics, etc. BERT [18], GPT-3 [9], PaLM [15], to name a few, are scaling to support a variety of tasks such as text generation, question-answering, text classification, arithmetic on numbers,

Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants

Gustavo Sandoval*, Hammond Pearce*, Teo Nys, Ramesh Karri, Siddharth Garg, Brendan Dolan-Gavitt
New York University

Abstract

Large Language Models (LLMs) such as OpenAI Codex are increasingly being used as AI-based coding assistants. Understanding the impact of these tools on developers' code is paramount, especially as recent work showed that LLMs may suggest cybersecurity vulnerabilities. We conduct a security-driven user study (N=58) to assess code written by student programmers when assisted by LLMs. Given the potential severity of low-level bugs as well as their relative frequency in real-world projects, we tasked participants with implementing a singly-linked 'shopping list' structure in C. Our results indicate that the security impact in this setting (low-level C with pointer and array manipulations) is small: AI-assisted users produce critical security bugs at a rate no greater than



Figure 1: What

Large Language Models for Code: Security Hardening and Adversarial Testing

Jingxuan He
ETH Zurich, Switzerland
jingxuan.he@inf.ethz.ch

Martin Vechev
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

ABSTRACT

Large language models (large LMs) are increasingly trained on massive codebases and used to generate code. However, LMs lack awareness of security and are found to frequently produce unsafe code. This work studies the security of LMs along two important axes: (i) security hardening, which aims to enhance LMs' reliability in generating secure code, and (ii) adversarial testing, which seeks to evaluate LMs' security at an adversarial standpoint. We address both of these by formulating a new security task called controlled code generation. The task is parametric and takes as input a binary property to guide the LM to generate secure or unsafe code, while

1 INTRODUCTION

After achieving great success in natural language [22, 30, 63, 73], large language models (large LMs) are extensively trained on the vast amount of available open-source code and used to generate functionally correct programs from user-provided prompts [18, 27, 34, 50, 56, 68, 76]. These models form the foundation of various commercial code completion engines [2, 3, 5, 8, 71]. In particular, the Codex model [25] powers GitHub Copilot [9]. According to GitHub's statistics, Copilot has been used by >1M developers and >5k businesses [31]. Many studies confirmed LMs' benefits in improving programming productivity [41, 65, 71, 72].

Although LMs excel in functional correctness, they may produce code with security issues [25, 27, 74]. An evaluation in [59] discovered that, in various security-relevant scenarios, 40% of Copilot-generated programs contain dangerous vulnerabilities. This evaluation was reused in [68], which found that other state-of-the-art LMs [34, 56, 68] have similarly concerning security level as Copilot. Another study in [43] found that in 16 out of 21 security-relevant cases, ChatGPT [4] generates code below minimal security standards. In practice, users can always reject or modify LM-suggested code, including any LM-generated vulnerabilities. The authors of the Copilot evaluation conducted a follow-up user study that considers such human interaction [65]. The study concluded that while LM-assistance provides productivity gain, it does not lead developers to produce significantly more security bugs. This finding reassures LMs' usefulness even in security-sensitive scenarios. However, considerable effort is still required to rule out vulnerabilities in LM-suggested code either manually during coding or through retrospective security analysis after coding.

Technologies → Machine learning, Security
Software and application security.

research highlights

Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions

By Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri

Abstract

There is burgeoning interest in designing AI-based systems to assist humans in designing computing systems, including tools that automatically generate computer code. The most notable of these comes in the form of the first self-described "AI pair programmer," GitHub Copilot, a language model trained over open source GitHub code. However, code often contains bugs—and we, given the vast quantity of unvetted code that Copilot has processed, it is certain that the language model will have learned from exploitable, buggy code. This raises concerns on the security of Copilot's code contributions. In this work, we systematically investigate the prevalence and conditions that can cause GitHub Copilot to recommend insecure code. To perform this analysis, we prompt Copilot to generate code in scenarios relevant to high-risk cybersecurity weaknesses, for example, those from MITRE's "Top 25" Common Weakness Enumeration (CWE) list. We explore Copilot's performance on three distinct code-generation axes—examining how it performs given diversity of weaknesses, diversity of prompts, and diversity of domains. In total, we produce 89 different scenarios for Copilot to complete, producing 1,689 programs. Of these, we found approximately 40% to be vulnerable.

el (LLM) trained on opensource code,¹ including "public code...with insecure coding patterns", thus giving rise to the potential for "synthesizing[d] code that contains these undesirable patterns."² Although prior research has evaluated the functionality of code generated by language models,^{3,4} there is no systematic examination of the security of ML-generated code. As GitHub Copilot is the largest and most capable such model currently available, it is important to understand: Are Copilot's suggestions commonly insecure? What is the prevalence of insecure generated code? What factors of the "context" yield generated code that is more or less secure?

We systematically experiment with Copilot to gain insights into these questions by designing scenarios for Copilot to complete and by analyzing the produced code for security weaknesses. As our corpus of well-defined weaknesses, we check Copilot completions for a subset of MITRE's Common Weakness Enumerations (CWEs), from their "2021 CWE Top 25 Most Dangerous Software Weaknesses"⁵ list. The AI's documentation recommends using "Copilot together with testing practices and security tools, as well as your own judgment." Our work attempts to characterize the tendency of Copilot to produce insecure code, giving a gauge for the amount of scrutiny a human developer might need to do for security issues.

Runtime Performance

- Page-load Latency vs Customer Bounce Rate ^[1]



- Latency vs Sales revenue ^[2]



- Large-scale systems track 99th percentile latency in μs ^[3]

[1] Google. Mobile Page Speed and Industry Benchmarks, 2017.

[2] Gigaspaces, Amazon Found Every 100ms of Latency Cost them 1% in Sales, 2023.

[3] Udayashankar S et al. Draconis: Network Accelerated Scheduling for Microsecond Scale Workloads, ACM SIGOPS European Conference on Computer Systems (EuroSys), 2024

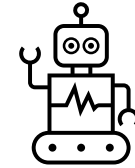
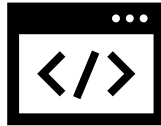
Our Contributions

- First study focused on runtime performance
 - Systems / Infrastructure Engineering

32 participants



Two C++ problems



+/- Copilot

- Sample RQ: *Is there a runtime performance difference in C++ code when using Copilot?*
 - **Spoiler Alert!**: On average, copilot-aided solutions were **15-27% slower** than their unaided counterparts.

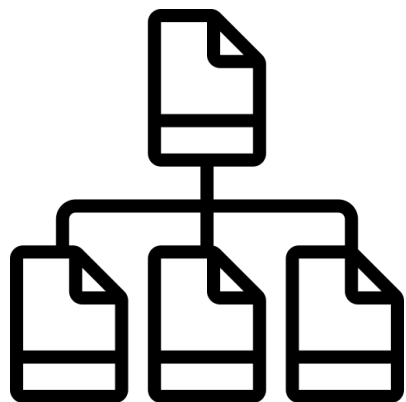
Outline

- Introduction
- Methodology
- Evaluation ~ RQs
- Takeaways

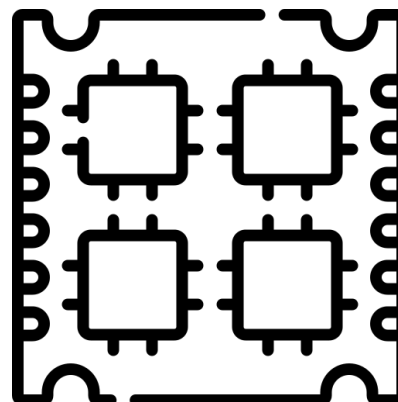
Methodology

- Participants asked to solve two problems
 - One with and another without Copilot assistance

32 Participants



Problem A: File I/O

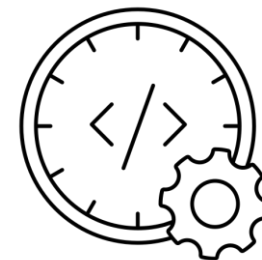


Problem B: Multi threading

Must be quickly solvable!



Must have multiple solutions with only performance differences!



UNIVERSITY OF
WATERLOO

Methodology

- Participants asked to solve two problems
 - One with and another without Copilot assistance
- Code Stubs

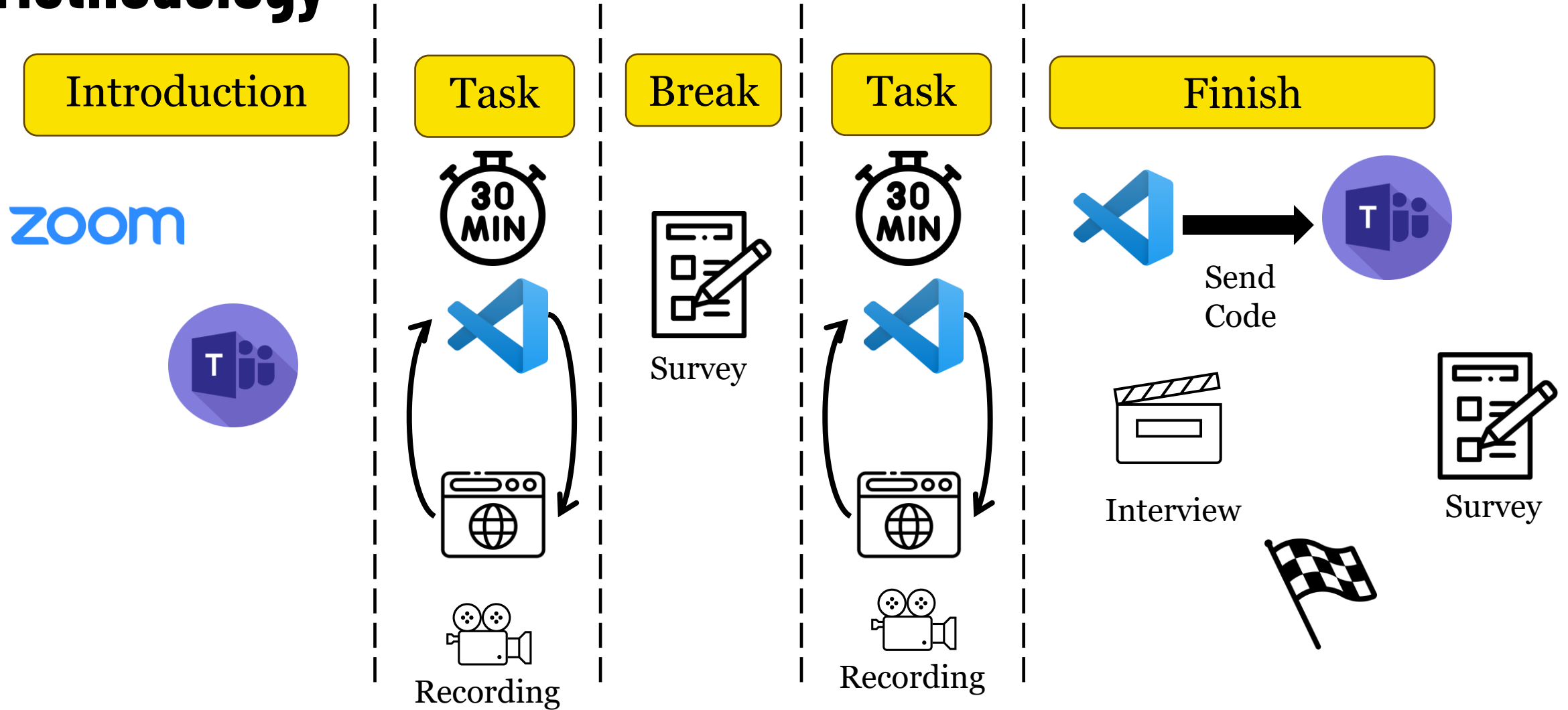


32 Participants



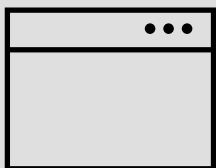
```
.....  
  
void primaryFunction() {  
    // YOUR CODE GOES HERE  
}  
  
.....  
  
int main() {  
    initialization();  
    primaryFunction();  
    sanityCheck();  
    return 0;  
}
```

Methodology

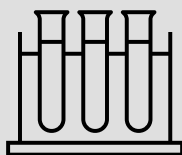


RESEARCH QUESTIONS

RQ 0: *Does Copilot influence program correctness?*



Code
Analysis



Test
Cases

Copilot leads developers to
produce correct code in most
cases!

Test Failure ✗ Compilation Failure ✗

#	partID	problem	mode	compiled	passed
1	P32 ✗	B	C	TRUE	FALSE
2	P30	B	NC	TRUE	FALSE
3	P23	A	NC	TRUE	NULL
4	P15 ✗	A	NC	FALSE	NULL
5	P15	B	C	FALSE	NULL
6	P7	A	NC	FALSE	NULL
7	P6	B	NC	TRUE	FALSE
8	P3	A	NC	TRUE	FALSE

TABLE OF INVALID RUNS

Without Copilot

A: 4/16

B: 2/16

With Copilot

A: 0/16

B: 2/16

RESEARCH QUESTIONS

RQ 1: *Is there a runtime performance difference in C++ code when using Copilot?*



Runtime Performance

Copilot-aided solutions possess worse runtime performance than unaided ones!

- 32 runs on 8-core Intel Xeon D-1548 @ 2 GHz
- Wilcoxon Rank Sum Test

Problem	Mode	Valid Runs	Mean	Median	Min	Max
A	C	16 x 32	34.86 s	34.85 s	33.82 s	36.02 s
A	NC	12 x 32	26.02 s	34.47 s	4.045 s	35.84 s
B	C	14 x 32	1898 ms	945.4 ms	612.1 ms	7356 ms
B	NC	14 x 32	1628 ms	943.9 ms	494.9 ms	6761 ms

TABLE OF RUNTIME PERFORMANCE

Problem A: Mean runtime with Copilot is **27% slower.**

Problem B: Mean runtime with Copilot is **15% slower.**

RESEARCH QUESTIONS

RQ 1: *Is there a runtime performance difference in C++ code when using Copilot?*



Runtime Performance

Copilot-aided solutions possess worse runtime performance than unaided ones!

- 32 runs on 8-core Intel Xeon D-1548 @ 2 GHz
- Wilcoxon Rank Sum Test

Problem	Mode	Valid Runs	Mean	Median	Min	Max
A	C	16 x 32	34.86 s	34.85 s	33.82 s	36.02 s
A	NC	12 x 32	26.02 s	34.47 s	4.045 s	35.84 s
B	C	14 x 32	1898 ms	945.4 ms	612.1 ms	7356 ms
B	NC	14 x 32	1628 ms	943.9 ms	494.9 ms	6761 ms

TABLE OF RUNTIME PERFORMANCE

Problem A: Fastest solution without Copilot is **8x faster** than with.

Same participant had a solution closer to median runtime when using CoPilot for Problem B!

RESEARCH QUESTIONS

RQ 1: *Is there a runtime performance difference in C++ code when using Copilot?*



Runtime Performance

Copilot-aided solutions possess worse runtime performance than unaided ones!

Problem A:
Mean runtime
with Copilot is
27% slower.

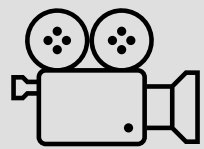
Problem B:
Mean runtime
with Copilot is
15% slower.

Problem A: Fastest solution without
Copilot is **8x faster** than with.

Same participant had a solution
closer to median runtime when using
CoPilot for Problem B!

RESEARCH QUESTIONS

RQ 2: *Does Copilot sway developers towards or away from solutions with faster runtime performance?*



Video
Analysis



Open
Coding



Participant
Survey

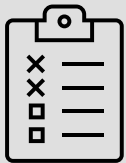
Copilot tends to sway developers
towards slower solutions!

P12 (B) when implementing an optimization:

*“Copilot didn’t understand me well; I just gave up
and wrote it myself.”*

RESEARCH QUESTIONS

Bonus: *How do participant demographics affect these results?*



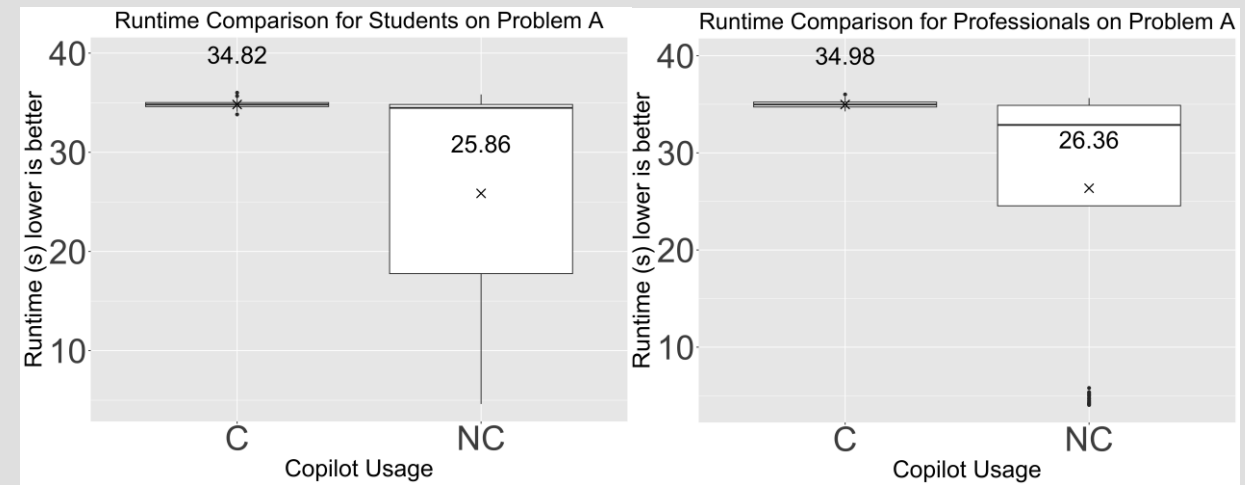
Participant
Survey



Runtime
Performance

Copilot-aided solutions are slower
regardless of participant
demographics!

Lower runtime is better.



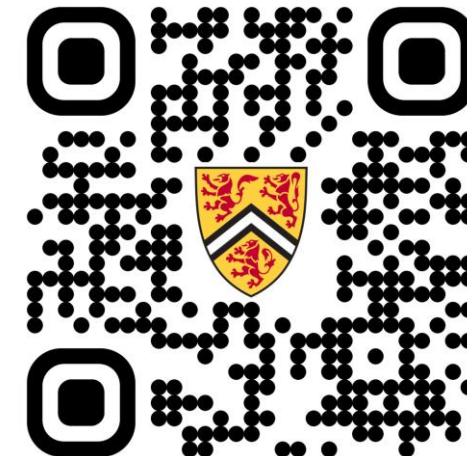
Students

Professionals

Problem A

Summary

- Coding assistants powered by LLMs are popular
 - Generated code needs to be carefully examined
- GitHub Copilot
 - Produces functionally correct code in most cases
 - Does not target better runtime performance and hinders developers trying to do so.
- Anonymized Participant Data / Scripts: [Artifact Link](#)



UNIVERSITY OF
WATERLOO

