

LMULUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHENMedia
Informatics
Group

IUIs for Software Development

Thomas Weber, Sven Mayer

Software Development Tools with AI

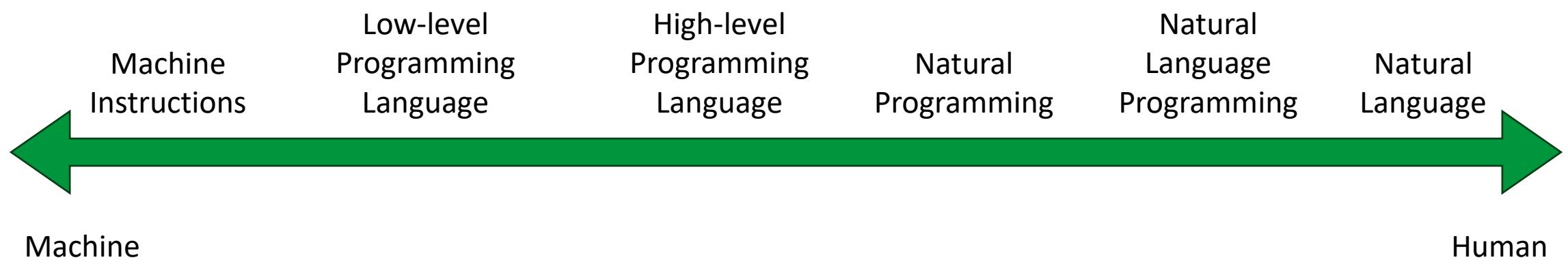
- Automatic bug detection
- Automatic testing
- Code optimization
- Documentation generation
- Automatic vulnerability detection
- Adaptive Dev Tools

Software Development Tools with AI

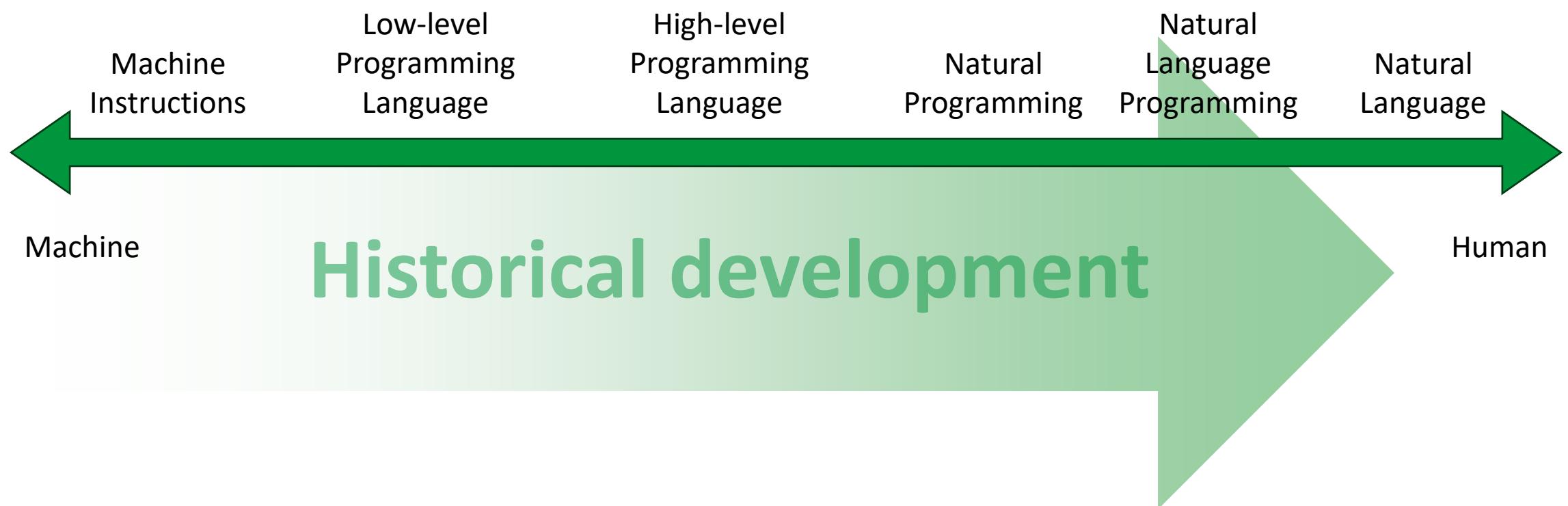
- Automatic bug detection
- Automatic testing
- Code optimization
- Documentation generation
- Automatic vulnerability detection
- Adaptive Dev Tools
 - How often do you use these tools?**
 - How often do you perform the tasks that these tools support?**

Programming with AI

Human-Machine-Interaction for Programming



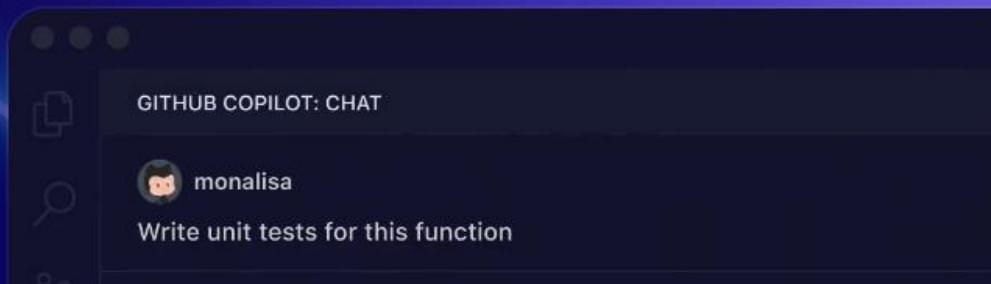
Human-Machine-Interaction for Programming



 GitHub Copilot

The world's most widely adopted AI developer tool

[Get started with GitHub Copilot >](#)



```
parse_expenses.py × addresses.rb × sentiments.ts ×  
1 import datetime  
2  
3 def parse_expenses (expenses_string):  
4     """Parse the list of expenses and return the list of triples (date, amount, currency)
```



// See Tabnine in action //

Plan > Create > Document > Test > Review > Explain > Maintain >

The screenshot shows the Tabnine AI Chat interface integrated into a code editor. On the left, a sidebar titled "TABNINE AI: CHAT" lists actions like "Explain code", "Write docstrings for code", "Fix code", "Generate tests for code", and "Explore codebase". A purple callout box highlights the "Explore codebase" button, with a hand cursor pointing at it. The main area displays a Java file named "Main.java" with code related to a password generator. The code includes imports for java.util.Scanner, a main method, and a Generator class. The Generator class has methods for printing colorful animals and banners. The code editor has a dark theme with syntax highlighting.

```
1 Main.java 2 ✎
1 import java.util.Scanner;
2
3 public class Main {
4
5     public static final Scanner keyboard = new Scanner(System.in);
6
7     public static void main(String[] args) {
8         printColorfulAnimal();
9         printColorfulBanner();
10        Generator generator = new Generator(keyboard);
11        generator.mainLoop();
12        keyboard.close();
13    }
14
15    private static void printColorfulBanner() {
16        System.out.println(x:"\u001B[36m");
17        System.out.println(x:"" );
18        System.out.println(x:"[ Best Password Generator v1.0 ]");
19        System.out.println(x:"" );
20        System.out.println(x:"\u001B[0m");
21    }
22
23    private static void printColorfulAnimal() {
24        System.out.println(x:"\u001B[36m");
25        System.out.println(x:" _-\\"/\\"/\\"- " );
26        System.out.println(x:" / - - \\" ");
27    }
28}
```

{ Code Generator }

Generate Code Online with AI 

Write a function in  that 

 Create
Standard 

Select a flavor

- Minimal 
- Standard 
- Documented 
- With Tests 

Your generated code will be shown here.

The AI Code Editor

Built to make you extraordinarily productive,
Cursor is the best way to code with AI.

[DOWNLOAD FOR WINDOWS](#)[WATCH DEMO](#)

A screenshot of the Cursor AI Code Editor interface. The main area shows a code editor with a dark theme. A tooltip box is open over some code, containing the text: "Implement the cleanup function for the transport stack. Do not make the upgrade listeners optional." Below the tooltip are buttons for "Accept", "Reject", and "Follow-up instructions...". The code editor shows several lines of Rust-like code, including:

```
72 pub(crate) struct TransportStack {
73     l4: ListenerEndpoint,
74     tls: Option<Arc<Acceptor>>,
75     // listeners sent from the old process for graceful upgrade
76     #[cfg(unix)]
77     upgrade_listeners: Option<ListenFds>,
78     upgrade_listeners: ListenFds,
```

To the right of the code editor is a "CHAT" window showing a message from the AI: "Could you make it easier to switch certificates in the transport listeners?". Below the message, the AI provides a response: "I'll help modify the code to make certificate switching more flexible. The main changes will be to enhance the `TlsAccept` trait and modify how certificates are handled in the `TlsSettings`. Here are the key changes:"

The "COMPOSER" window at the bottom shows a partial code snippet starting with "mod.rs" and "impl TransportStack {".



• Home Enterprise Pricing

About us ↗ Careers ↗ Blog ↗ Contact ↗

Login ↗

Get started



Built by Cognition

Devin
is a collaborative
AI teammate

Built to help ambitious engineering
teams achieve more.

Scroll down ↓



GitHub Spark

Can we enable anyone to create or adapt software for themselves, using AI and a fully-managed runtime?

GITHUB NEXT

WHAT'S IT FOR?

Building and sharing personalized micro apps (“sparks”)

SHARE



STAGE

TECHNICAL PREVIEW

WHO MADE IT?

 Devon Rifkin

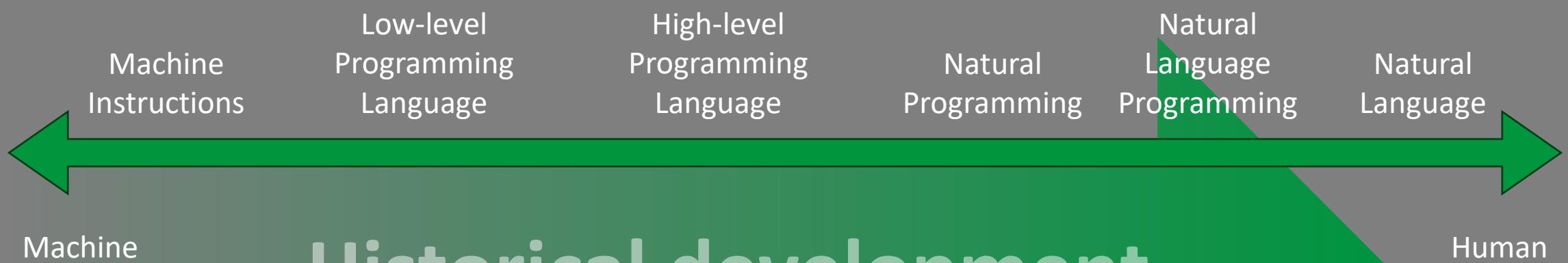
 Terkel Gjervig Nielsen

 Cole Bemis

 Alice Li

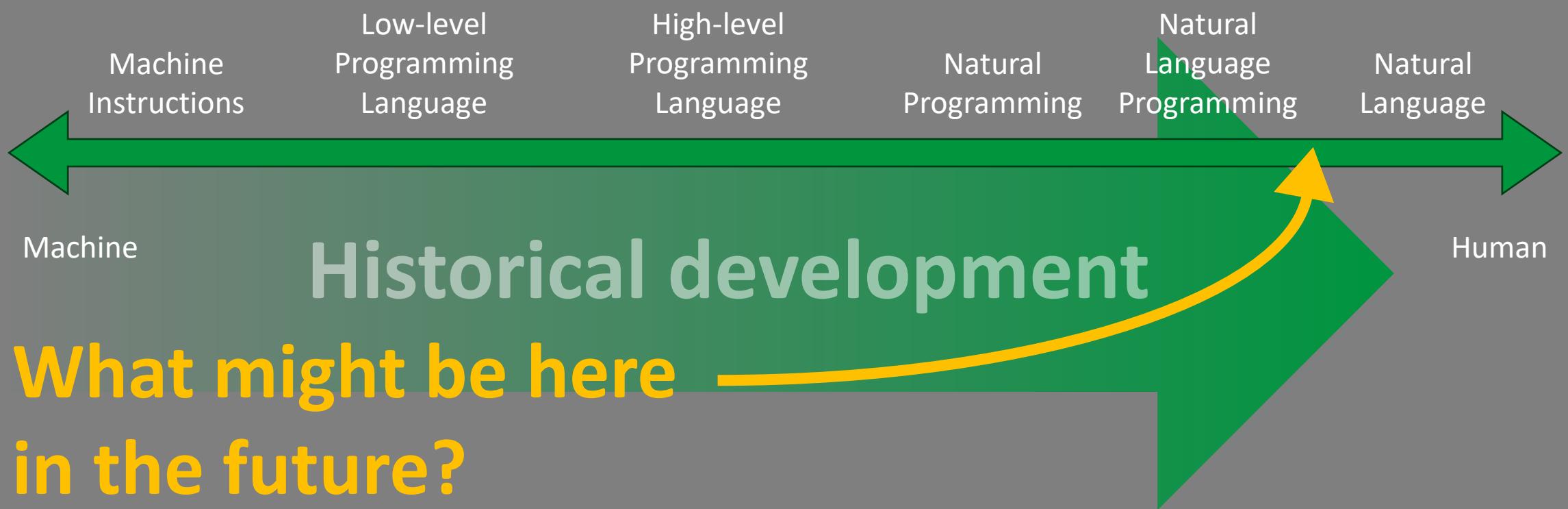
<https://youtu.be/zB-izOTcQ6s?t=24>

Human-Machine-Interaction for Programming



Do those on the right replace those on the left?

Human-Machine-Interaction for Programming



Leaderboard

Model	% Resolved	Org	Date	Logs	Trajs	Site
NEW 🏅 Amazon Q Developer Agent (v20241202-dev)	54.80	aws	2024-12-02	✓	✓	🔗
NEW 🏅 devlo	54.20	@@	2024-11-08	✓	✓	🔗
🏅 🤖 ✅ OpenHands + CodeAct v2.1 (claude-3-5-sonnet-20241022)	53.00	🤖	2024-10-29	✓	✓	🔗
NEW Engine Labs (2024-11-25)	51.80	蝘	2024-11-25	✓	✓	🔗
NEW 🤖 Agentless-1.5 + GPT 4o (claude-3-5-sonnet-20241022)	50.80	💡	2024-12-02	✓	✓	🔗
Solver (2024-10-28)	50.00	[\$]	2024-10-28	✓	✓	🔗
NEW Bytedance MarsCode Agent	50.00	Ⓜ	2024-11-25	✓	✓	🔗
NEW nFactorial (2024-11-05)	49.20	智力	2024-11-05	✓	✓	🔗
Tools + Claude 3.5 Sonnet (2024-10-22)	49.00	AI	2024-10-22	✓	✓	🔗
🏅 🤖 ✅ Composio SWE-Kit (2024-10-25)	48.60	⚡	2024-10-25	✓	✓	🔗
NEW 🤖 ✅ AppMap Navie v2	47.20	˄	2024-11-06	✓	✓	🔗
Emergent E1 (v2024-10-12)	46.60	ⓔ	2024-10-23	✓	✓	🔗
NEW 🤖 AutoCodeRover-v2.0 (Claude-3.5-Sonnet-20241022)	46.20	🤖	2024-11-08	✓	✓	🔗
Solver (2024-09-12)	45.40	[\$]	2024-09-24	✓	✓	🔗
Gru(2024-08-24)	45.20	⊕	2024-08-24	✓	✓	🔗
Solver (2024-09-12)	43.60	[\$]	2024-09-20	✓	✓	🔗
nFactorial (2024-10-30)	41.60	智力	2024-10-30	✓	✓	🔗
NEW Nebius AI Qwen 2.5 72B Generator + LLama 3.1 70B Critic	40.60	⠇	2024-11-13	✓	✓	🔗
Tools + Claude 3.5 Haiku	40.60	AI	2024-10-22	✓	✓	🔗
Honeycomb	40.60	ঢ়	2024-08-20	✓	✓	🔗
🏅 🤖 Composio SWEkit + Claude 3.5 Sonnet (2024-10-16)	40.60	⚡	2024-10-16	✓	✓	🔗
EPAM AI/Run Developer Agent v20241029 + Anthropic Claude 3.5 Sonnet	39.60	<epam>	2024-10-29	✓	✓	🔗
Amazon Q Developer Agent (v20241029-dev)	38.80	aws	2024-10-29	/	/	🔗

SWE-bench **Lite** is a subset of SWE-bench that's been curated to make evaluation less costly and more accessible [Post].

SWE-bench **Verified** is a human annotator filtered subset that has been deemed to have a ceiling of 100% resolution rate [Post].

- The **% Resolved** metric refers to the percentage of SWE-bench instances (2294 for test, 500 for verified, 300 for lite) that were resolved by the model.
- **Checked** indicates that we, the SWE-bench team, received access to the system and were able to reproduce the patch generations.

How much does it cost to replace one human with AI?

Typical SWE salary: \$220,000

Benefits, taxes, free breakfast, lunch, dinner, snacks,
masseuse, shuttle bus, on-site doctor, bowling alley, ...

\$92,000

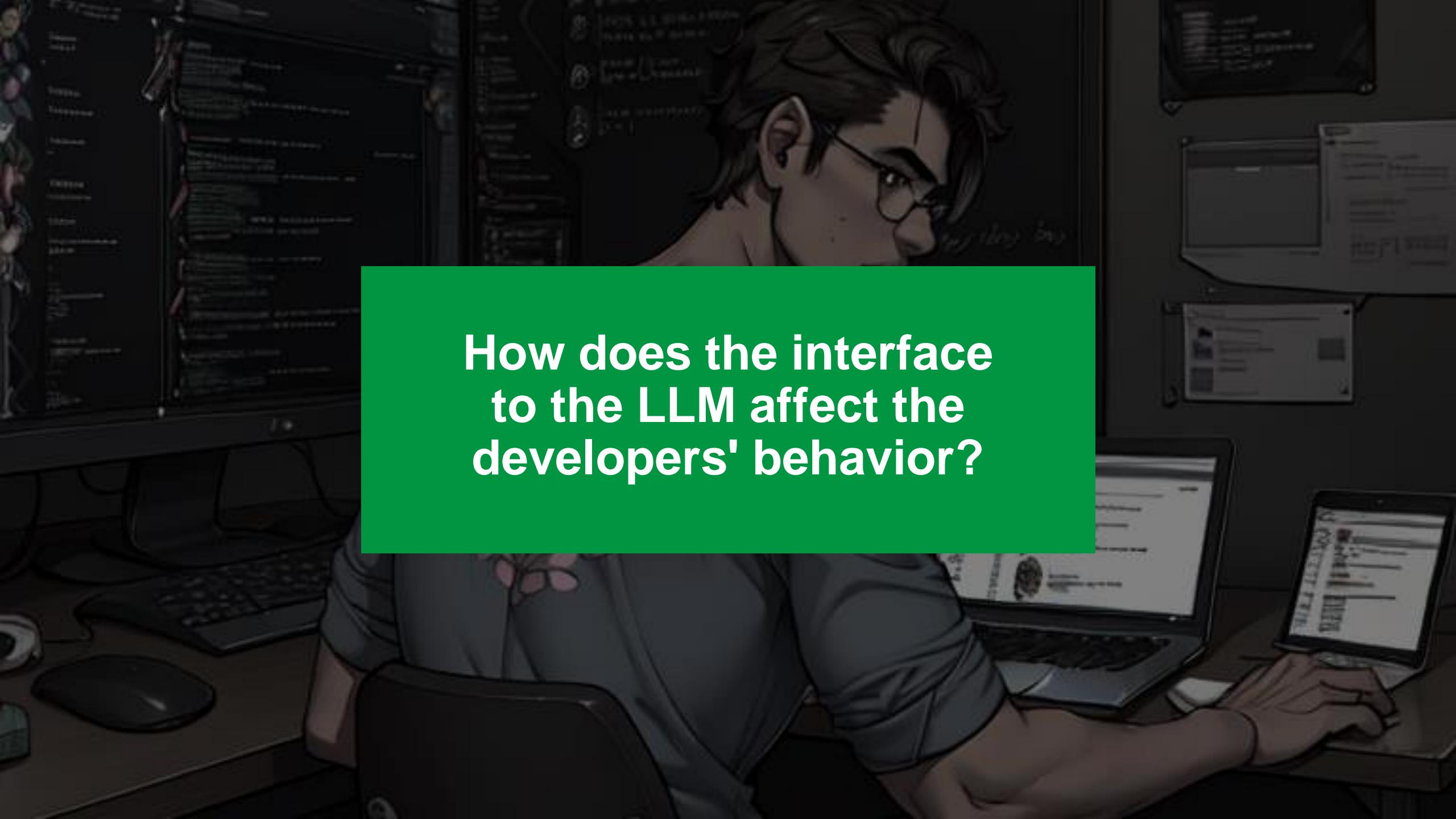
Total: \$312,000

Number of working days per year: 260

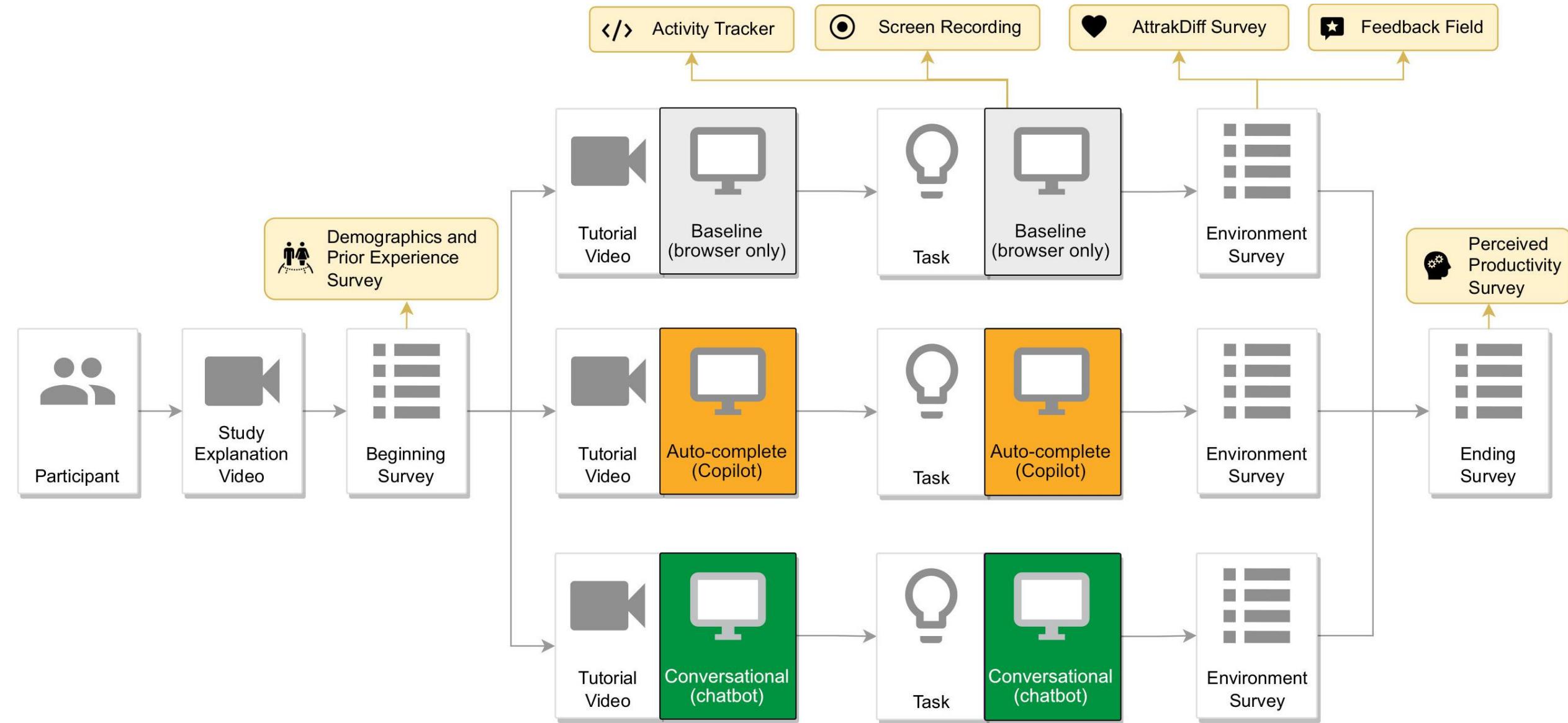
Total cost for one-human-SWE-day: \$1200



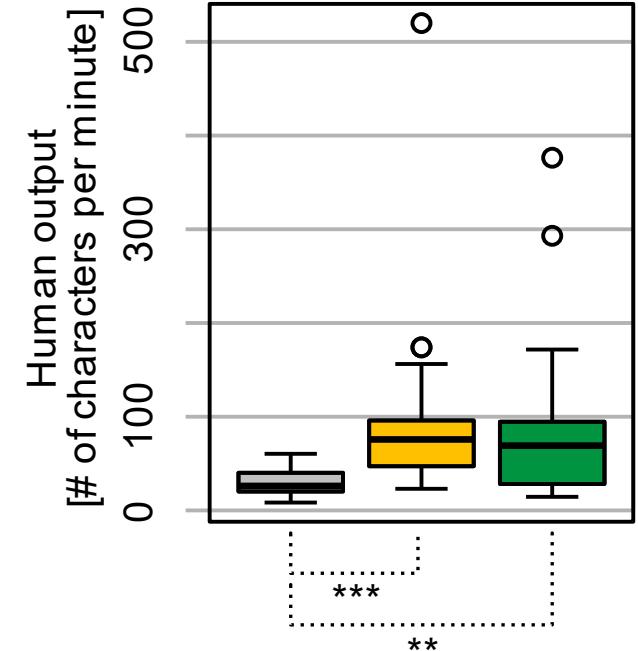
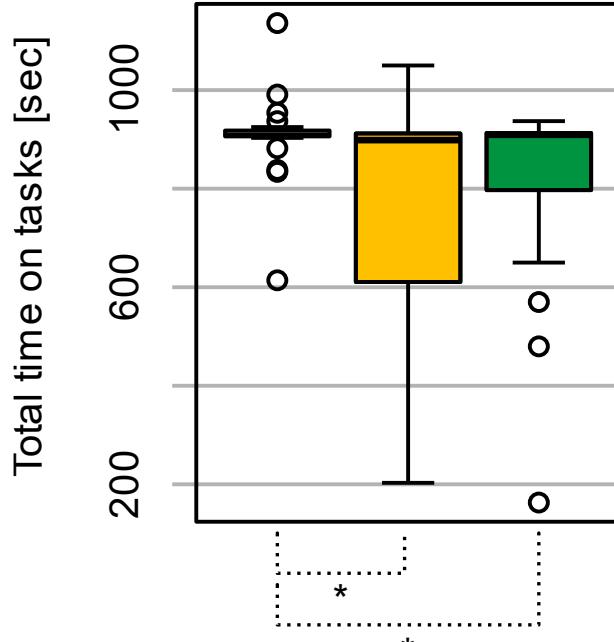
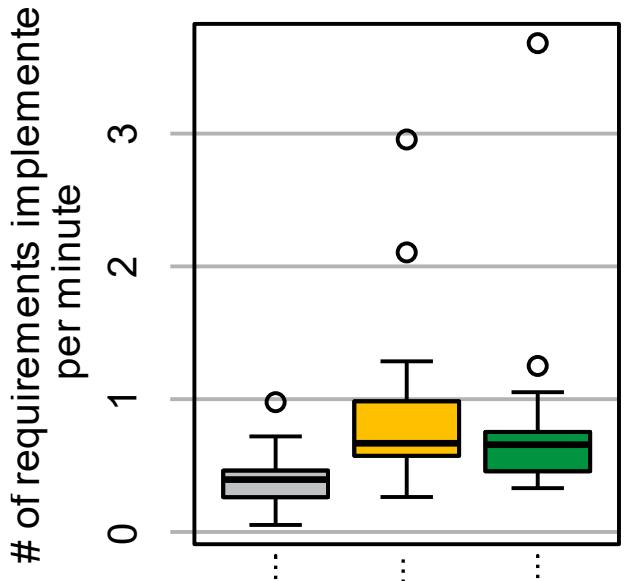
18

A developer with glasses is shown from the side, looking down at a computer screen. They are wearing a light blue t-shirt with a floral pattern on the sleeve. The background is dark, and there are multiple computer monitors displaying code and interface snippets.

**How does the interface
to the LLM affect the
developers' behavior?**



Results



Baseline
(browser only)

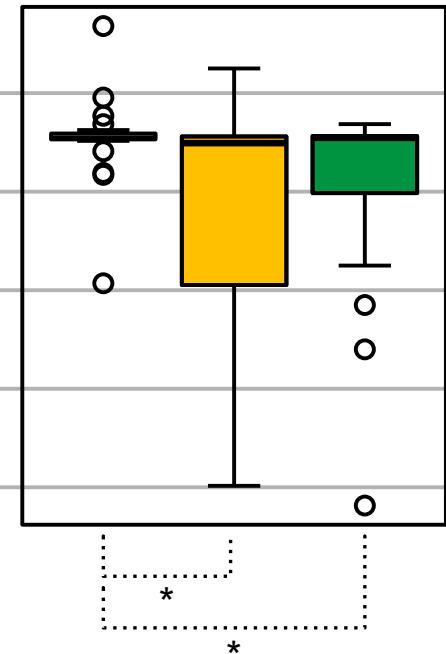
Auto-complete
(Copilot)

Conversational
(chatbot)

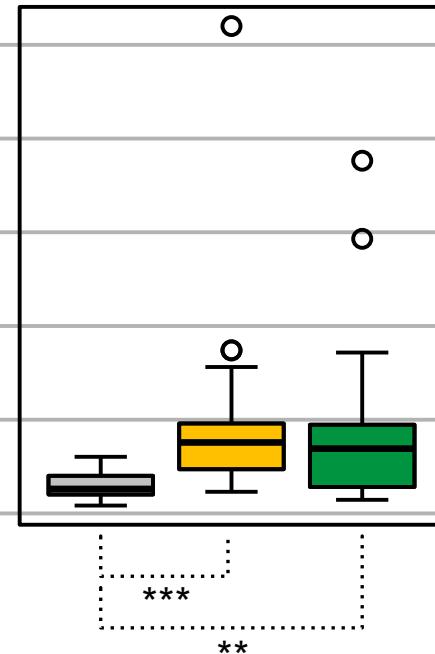
% of total code created by human

Results

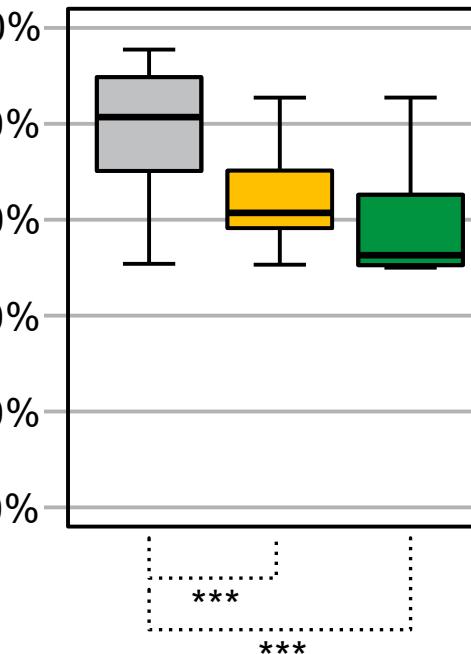
Total time on tasks [sec]



Human output [# of characters per minute]



% of total code created by human

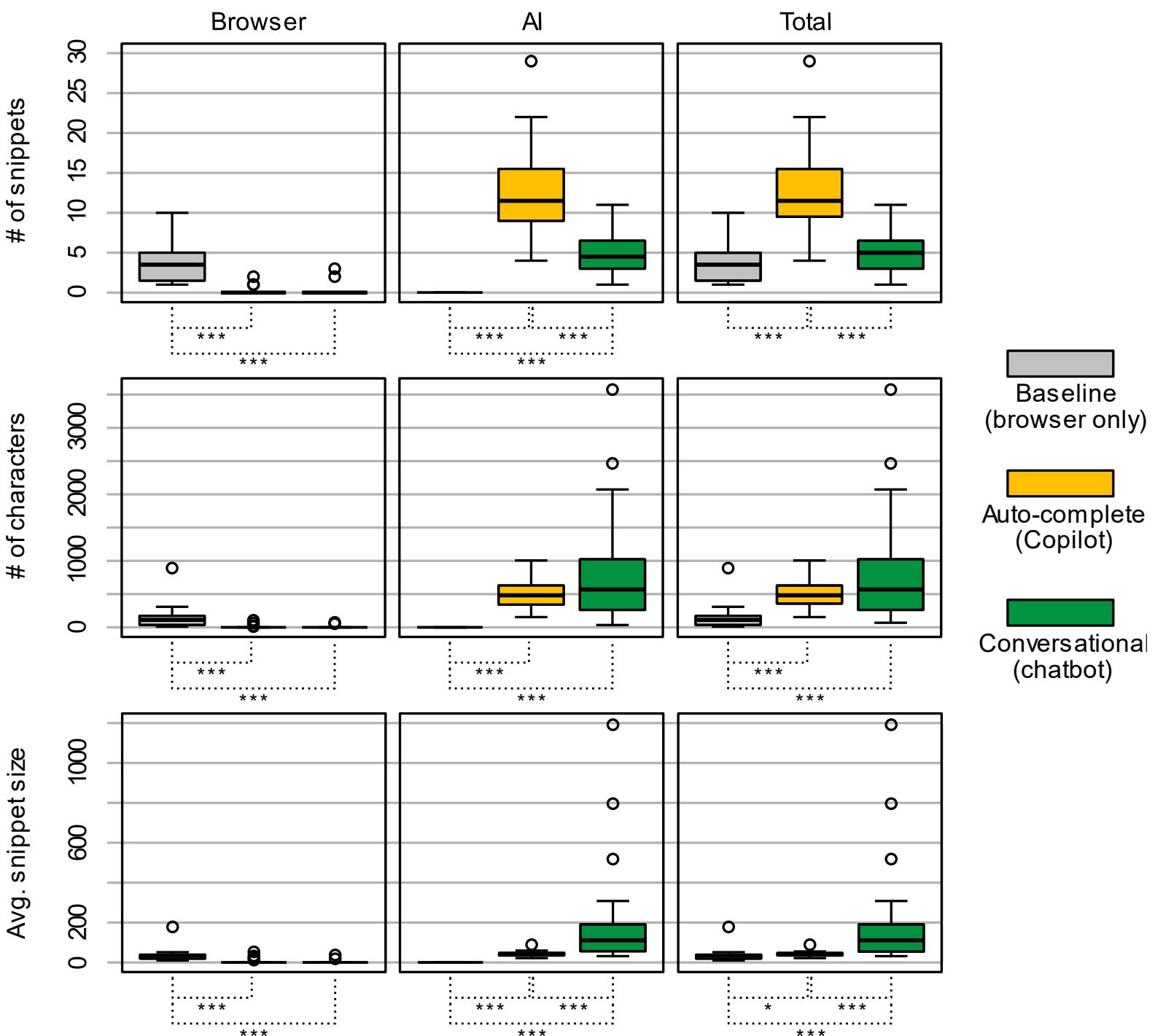


Baseline
(browser only)

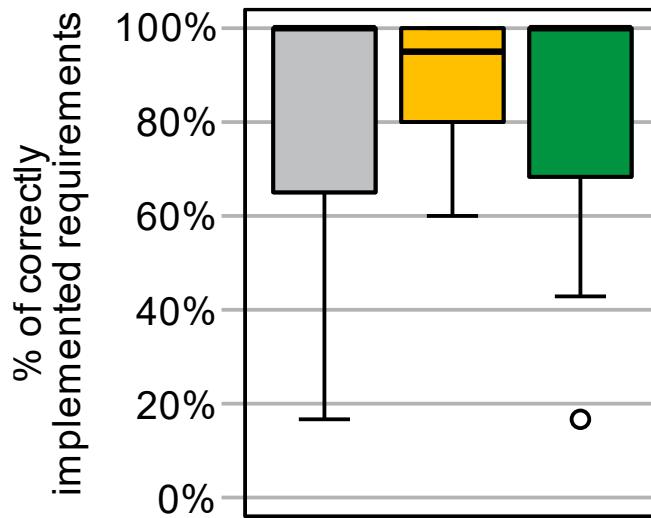
Auto-complete
(Copilot)

Conversational
(chatbot)

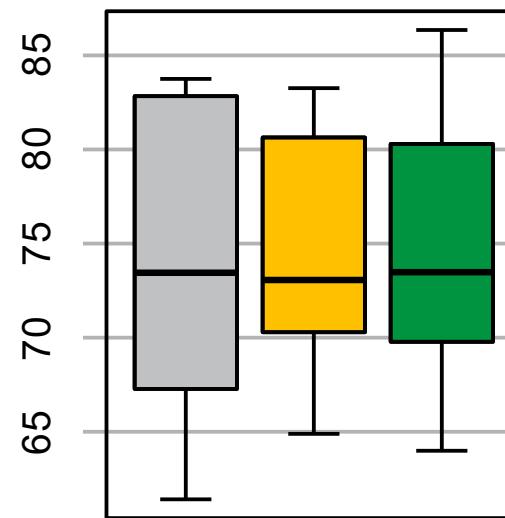
Results



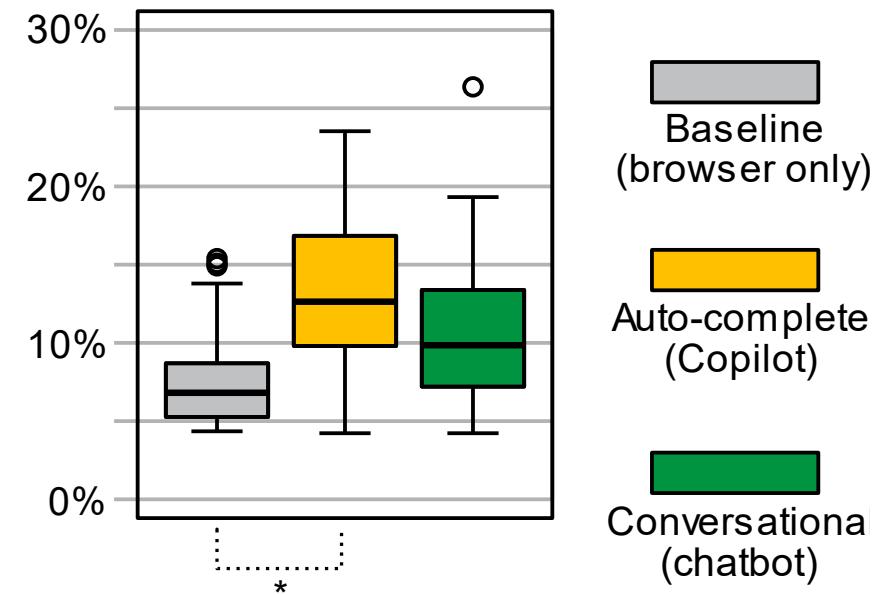
Results



Maintainability Index



Percentage of Comments



Legend:

- Baseline (browser only)
- Auto-complete (Copilot)
- Conversational (chatbot)

Consequences

- Human responsibilities shift
 - Coding → Steering, review
 - New/change in skills necessary
- Coding quality averages out (with current models)
- Level of abstraction increases
- Code for AI not for humans

AI Coders Are among Us: Rethinking Programming Language Grammar towards Efficient Code Generation

Zhensu Sun
Singapore Management University
Singapore
zssun@smu.edu.sg

Xiaoning Du*
Monash University
Australia
xiaoning.du@monash.edu

Zhou Yang
Singapore Management University
Singapore
zyang@smu.edu.sg

Li Li
Beihang University
China
lilicoding@ieee.org

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Abstract

Artificial Intelligence (AI) models have emerged as another important audience for programming languages alongside humans and machines, as we enter the era of large language models (LLMs). LLMs can now perform well in coding competitions and even write programs like developers to solve various tasks, including mathematical problems. However, the grammar and layout of current programs are designed to cater the needs of human developers – with many grammar tokens and formatting tokens being used to make the code easier for humans to read. While this is helpful, such a design adds unnecessary computational work for LLMs, as each token they either use or produce consumes computational resources.

To improve inference efficiency and reduce computational costs, we propose the concept of *AI-oriented grammar*. This aims to represent code in a way that better suits the working mechanism of AI models. Code written with AI-oriented grammar discards formats and uses a minimum number of tokens to convey code semantics effectively. To demonstrate the feasibility of this concept, we explore and implement the first AI-oriented grammar for Python, named Simple Python (SIMPy). SIMPy is crafted by revising the original Python grammar through a series of heuristic rules. Programs written in SIMPy maintain identical Abstract Syntax Tree (AST) structures to those in standard Python. This allows for not only exe-

models can maintain or even improve their performance when using SIMPy instead of Python for these tasks. With these promising results, we call for further contributions to the development of AI-oriented program grammar within our community.

CCS Concepts

- Computing methodologies → Artificial intelligence; Philosophical/theoretical foundations of artificial intelligence;
- Theory of computation → Grammars and context-free languages.

Keywords

Code Generation, Programming Language, Large Language Model

ACM Reference Format:

Zhensu Sun, Xiaoning Du, Zhou Yang, Li Li, and David Lo. 2024. AI Coders Are among Us: Rethinking Programming Language Grammar towards Efficient Code Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3650212.3680347>

1 Introduction

High-level programming languages like Python, Java, and C++ are designed with two types of audiences in mind [6]: machines that

AI Coders Are among Us: Rethinking Programming Language Grammar towards Efficient Code Generation

Zhensu Sun

Singapore Management University

Xiaoning Du*

Monash University

Zhou Yang

Singapore Management University

However, the grammar and layout of current programs are designed to cater the needs of human developers – with many grammar tokens and formatting tokens being used to make the code easier for humans to read

(...)

To improve inference efficiency and reduce computational costs, we propose the concept of **AI-oriented grammar**. This aims to represent code in a way that **better suits the working mechanism of AI models**

Ab

Arti-

tant

mac-

LLM

pro-

ema-

pro-

with

mak-

such

each

resources.

To improve inference efficiency and reduce computational costs, we propose the concept of *AI-oriented grammar*. This aims to represent code in a way that better suits the working mechanism of AI models. Code written with AI-oriented grammar discards formats and uses a minimum number of tokens to convey code semantics effectively. To demonstrate the feasibility of this concept, we explore and implement the first AI-oriented grammar for Python, named Simple Python (SIMPy). SIMPy is crafted by revising the original Python grammar through a series of heuristic rules. Programs written in SIMPy maintain identical Abstract Syntax Tree (AST) structures to those in standard Python. This allows for not only exe-

Code Generation, Programming Language, Large Language Model

ACM Reference Format:

Zhensu Sun, Xiaoning Du, Zhou Yang, Li Li, and David Lo. 2024. AI Coders Are among Us: Rethinking Programming Language Grammar towards Efficient Code Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3650212.3680347>

1 Introduction

High-level programming languages like Python, Java, and C++ are designed with two types of audiences in mind [6]: machines that

AI Coders Are among Us: Rethinking Programming Language Grammar towards Efficient Code Generation

Zhensu Sun

Singapore Management University

Xiaoning Du*

Monash University

Zhou Yang

Singapore Management University

However, the grammar and layout of current programs are designed to cater the needs of human developers – with many grammar tokens and formatting tokens being used to make the code easier for humans to read

(...)

To improve inference efficiency and reduce computational costs, we propose the concept of **AI-oriented grammar**. This aims to represent code in a way that **better suits the working mechanism of AI models**

Abstract. Artificial intelligence (AI) has revolutionized various fields, including software development. One key challenge is how to effectively utilize AI in the context of programming languages. This paper proposes a novel approach called AI-oriented grammar, which aims to represent code in a way that better suits the working mechanism of AI models. By discarding unnecessary tokens and using a minimum number of tokens to convey code semantics effectively, this approach can significantly improve inference efficiency and reduce computational costs.

To improve inference efficiency and reduce computational costs, we propose the concept of *AI-oriented grammar*. This aims to represent code in a way that better suits the working mechanism of AI models. Code written with AI-oriented grammar discards formats and uses a minimum number of tokens to convey code semantics effectively. To demonstrate the feasibility of this concept, we explore and implement the first AI-oriented grammar for Python, named Simple Python (SIMPy). SIMPy is crafted by revising the original Python grammar through a series of heuristic rules. Programs written in SIMPy maintain identical Abstract Syntax Tree (AST) structures to those in standard Python. This allows for not only exe-

Code Generation, Programming Language, Large Language Model

ACM Reference Format:

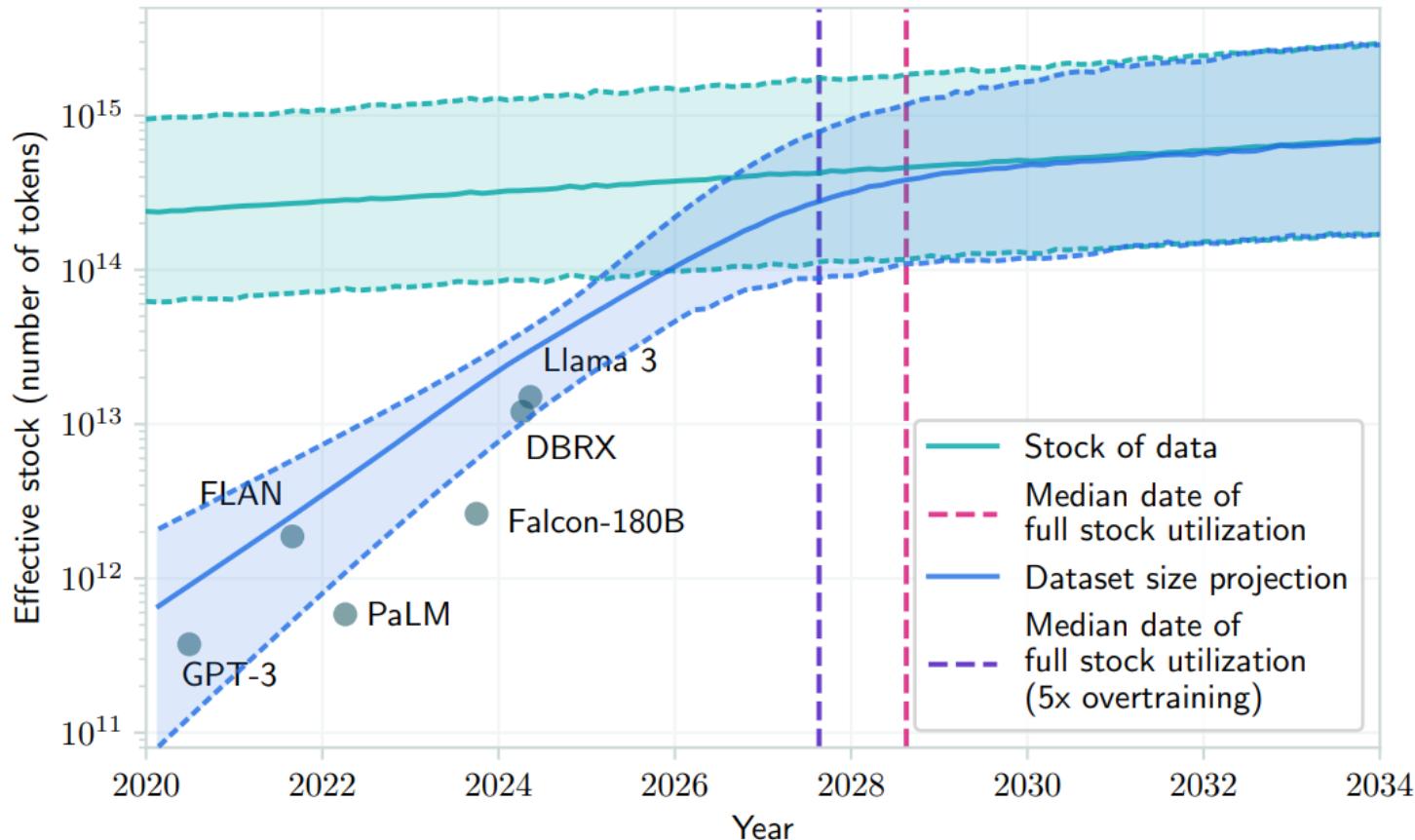
Zhensu Sun, Xiaoning Du, Zhou Yang, Li Li, and David Lo. 2024. AI Coders Are among Us: Rethinking Programming Language Grammar towards Efficient Code Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3650212.3680347>

1 Introduction

High-level programming languages like Python, Java, and C++ are designed with two types of audiences in mind [6]: machines that

Challenges

Challenges – Training Data



Challenges

- Unclear requirements
- Unclear how well LLMs can determine code quality [1]
- What if the code does not work? → Debugging
- Lack of reflection, learning, and problem-solving
- Licensed code in the training data and output
- Security, Reliability, etc.

[1] Simões, I. R. D. S., & Venson, E. (2024). Evaluating Source Code Quality with Large Language Models: a comparative study. *arXiv preprint arXiv:2408.07082*.

Conclusion

Questions?

Contact

Thomas Weber
LMU Munich
thomas.weber@ifi.lmu.de