ICPC 2025 The 33rd IEEE/ACM International Conference on Program Comprehension

Ottawa, ON, Canada – April 27 – 28, 2025

Theories of Program Comprehension in the Age of LLMs

Thomas LaToza

Developer Experience Design Laboratory

GEORGE MASON UNIVERSITY

Vibe coding with an LLM

+ Create App
Home
Apps
Deployments
Usage
Teams
Explore Replit
Bounties
Templates
Learn
Documentation

Hi Thomas, what do you want to make?

Describe an app or site you want to create...

App type: Auto
Start building
AI chat
Book scanner
Stock analysis
New

Agent just got better!

We've granted you 10 free checkpoints so you can try it out. Click to learn more.

Your Starter Plan

Free Apps
1/3 created

Agent Checkpoints
0/10 used | Expire 5/3/2025

Upgrade to Replit Core

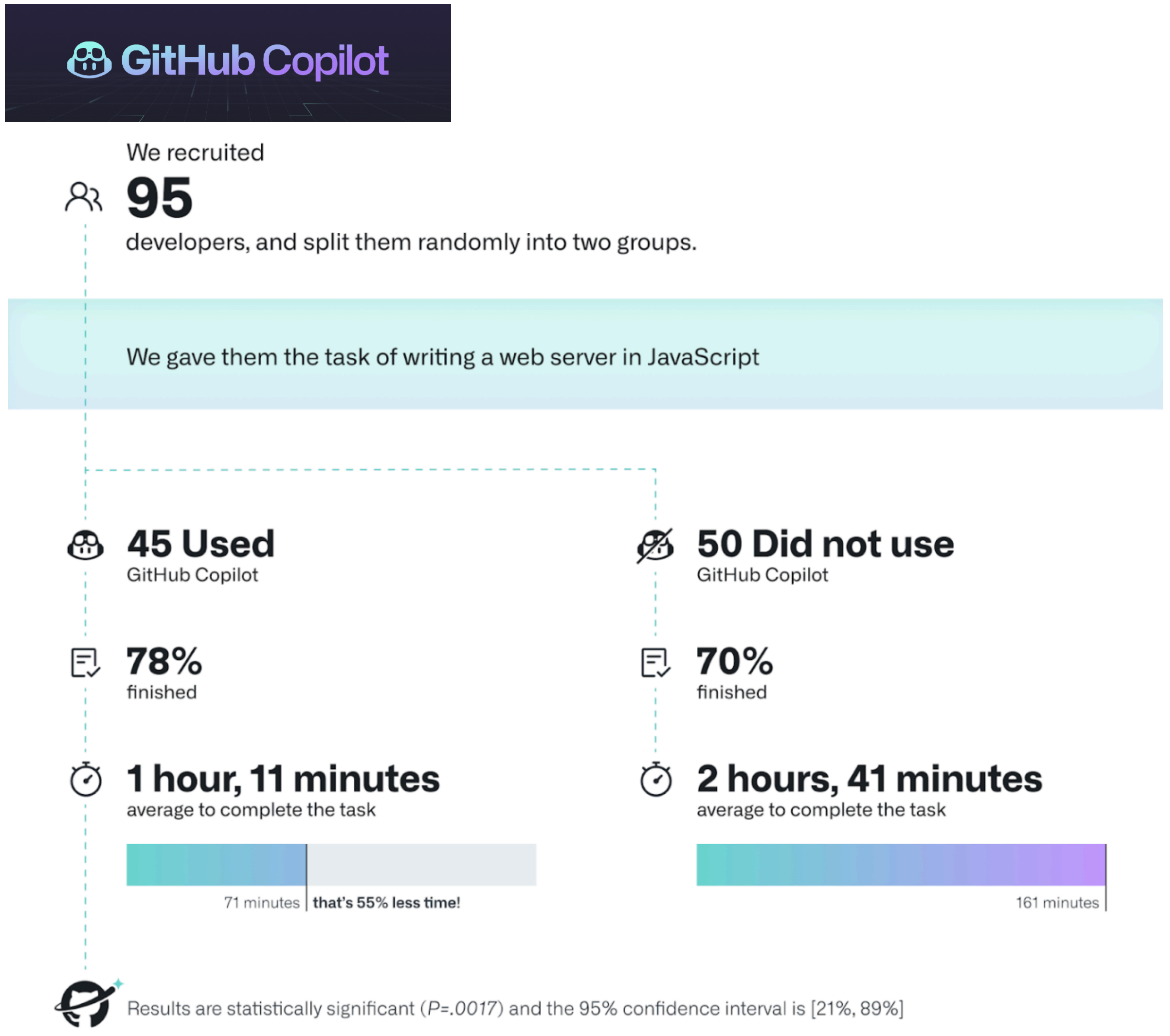
Your recent Apps

View All →

Programming is changing **fundamentally**.

Future of programming will be less about coding and more about **program comprehension**

- How can developers understand the code LLMs generate?
- How much understanding is still necessary?
- How do developers figure out what's wrong when it doesn't work?


GitHub Copilot

We recruited **95** developers, and split them randomly into two groups.

We gave them the task of writing a web server in JavaScript

45 Used
GitHub Copilot

78%
finished

1 hour, 11 minutes
average to complete the task

71 minutes | **that's 55% less time!**

50 Did not use
GitHub Copilot

70%
finished

2 hours, 41 minutes
average to complete the task

161 minutes

Results are statistically significant (*P=.0017*) and the 95% confidence interval is [21%, 89%]

"Today, more than a quarter of all new code at Google is generated by AI, then reviewed and accepted by engineers. This helps our engineers do more and move faster."

Sundar Pichai
CEO, Google
10/30/2024

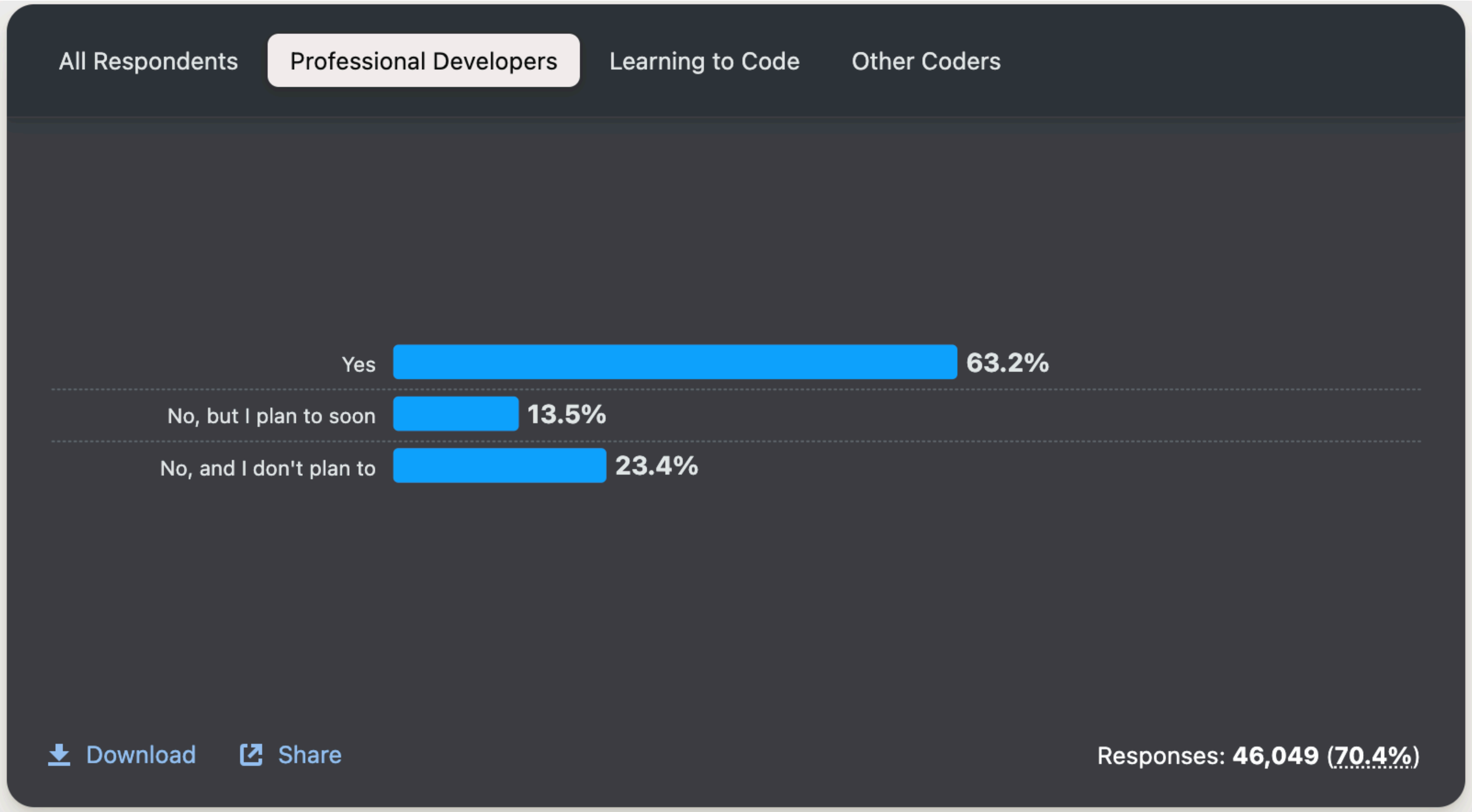
https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

stackoverflow 2024 Developer Survey

AI tools in the development process

76% of all respondents are using or are planning to use AI tools in their development process this year, an increase from last year (70%). Many more developers are currently using AI tools this year, too (62% vs. 44%).

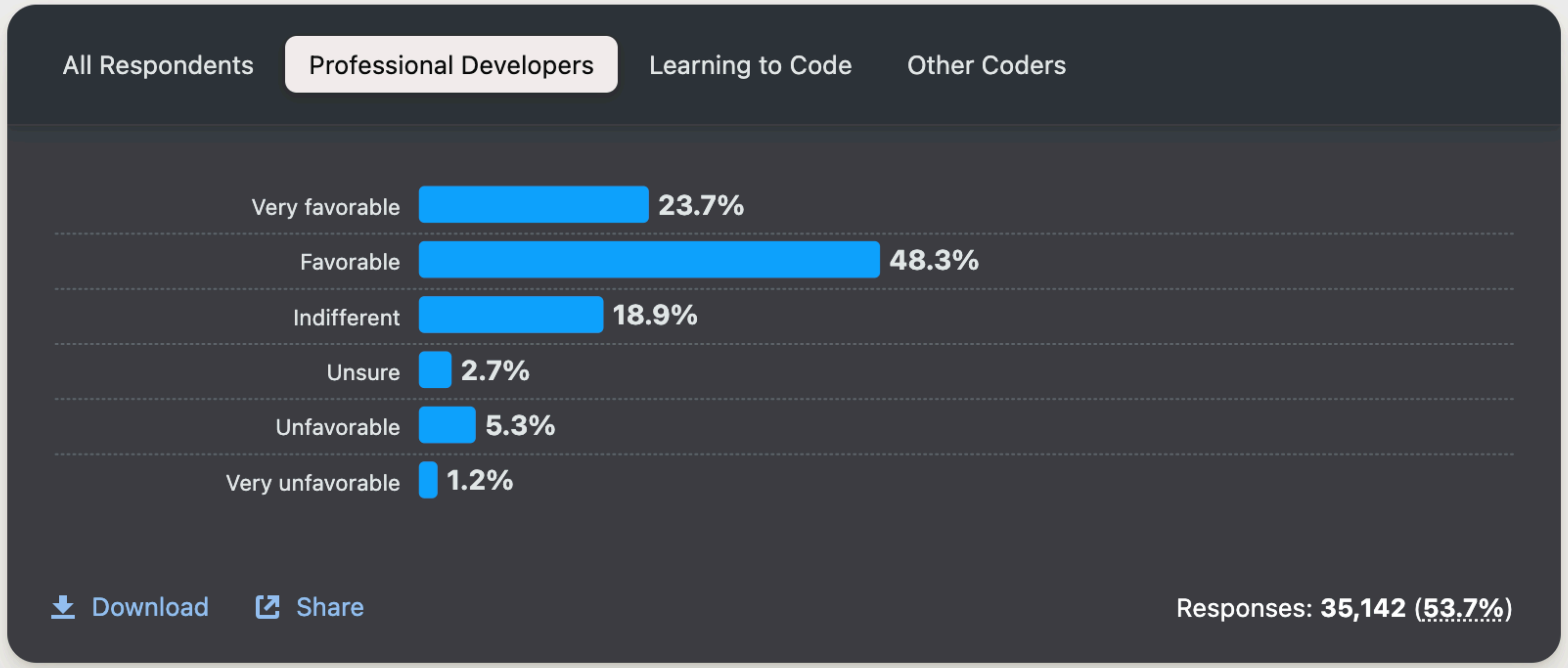
Do you currently use AI tools in your development process? *



AI tool sentiment

72% of all respondents are favorable or very favorable of AI tools for development. This is lower than last year's favorability of 77%; a decline in favorability could be due to disappointing results from usage.

How favorable is your stance on using AI tools as part of your development workflow?



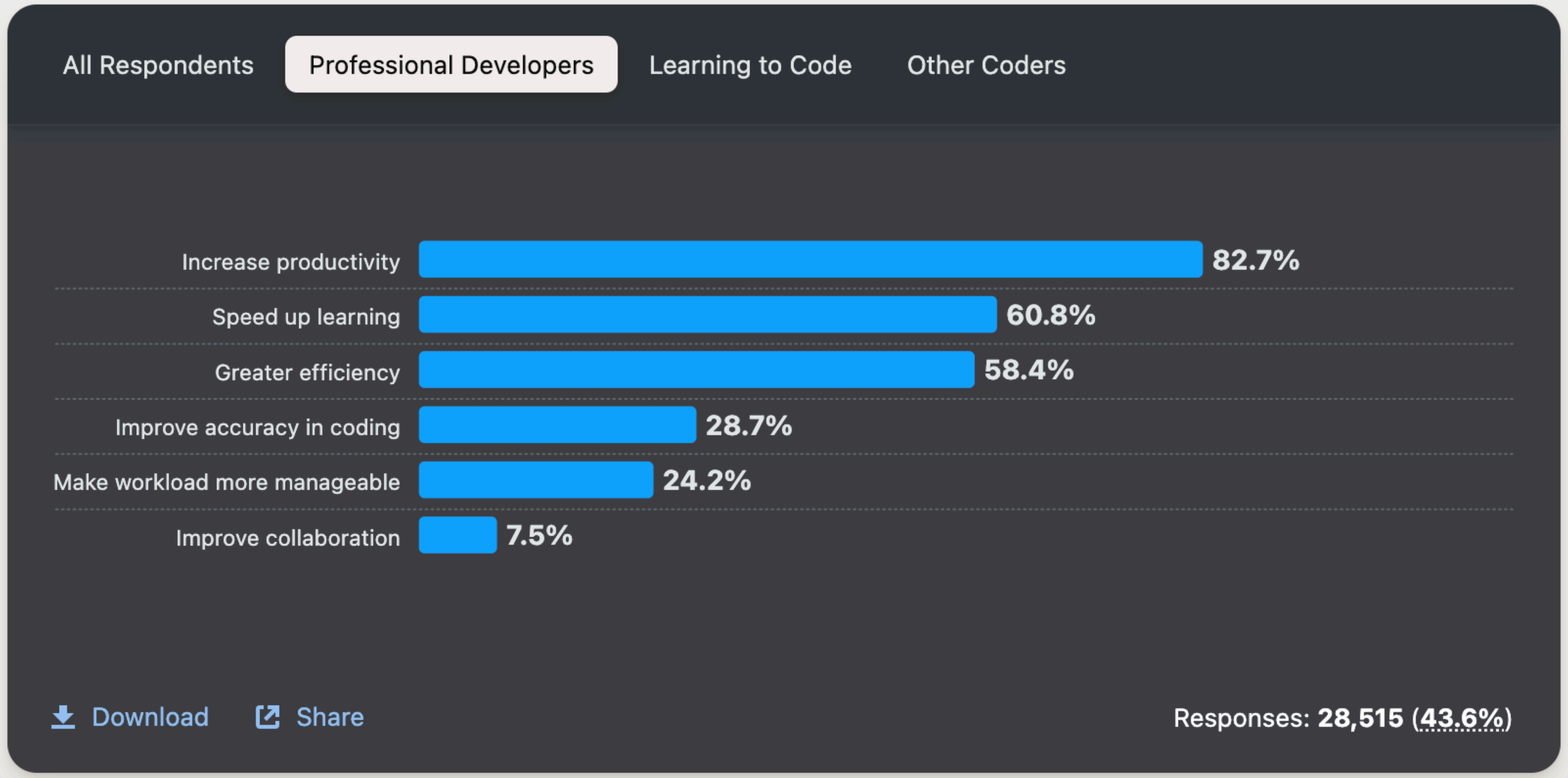
https://survey.stackoverflow.co/2024/ai

stackoverflow 2024 Developer Survey

Benefits of AI tools

81% agree increasing productivity is the biggest benefit that developers identify for AI tools. Speeding up learning is seen as a bigger benefit to developers learning to code (71%) compared to professional developers (61%).

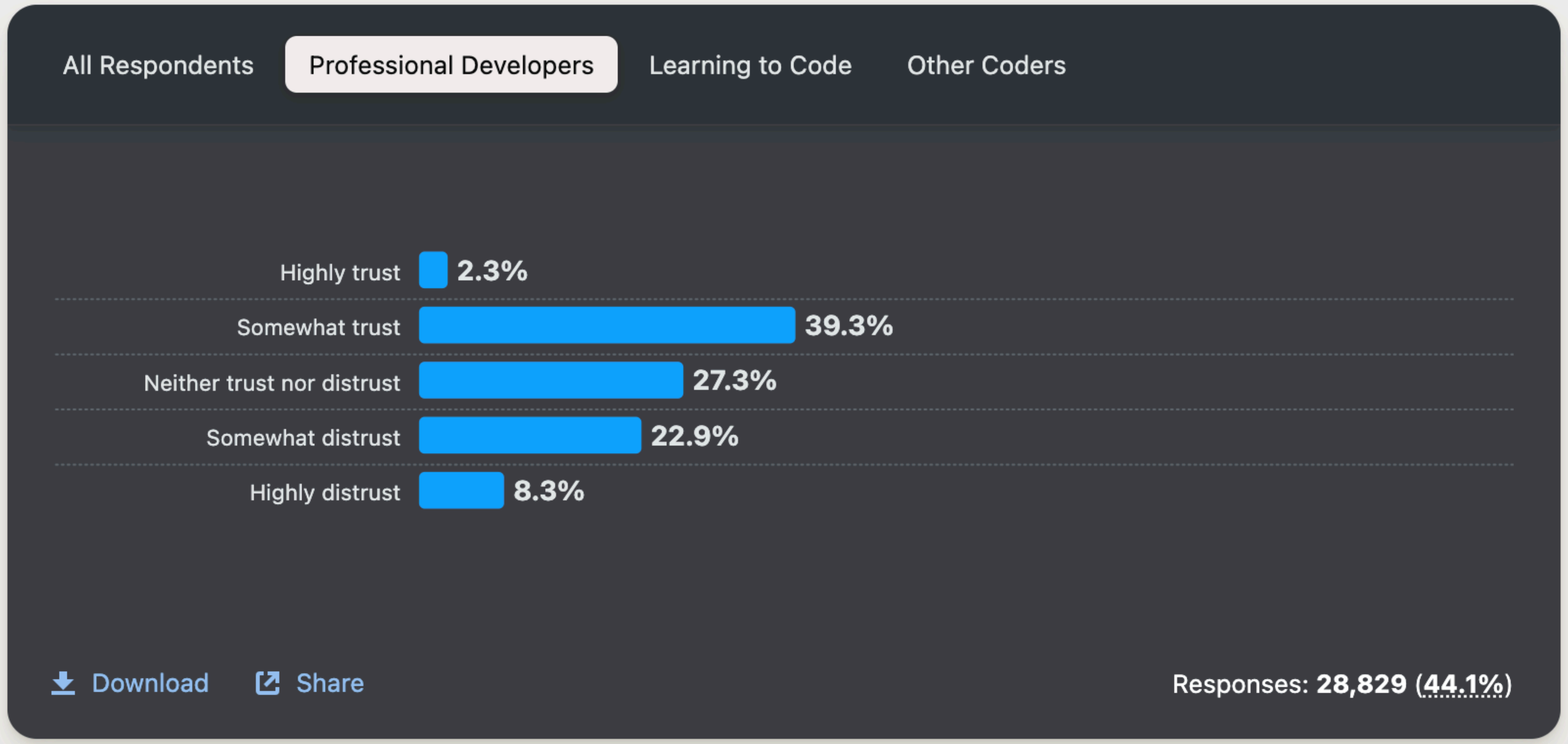
For the AI tools you use as part of your development workflow, what are the MOST important benefits you are hoping to achieve? Please check all that apply.



Accuracy of AI tools

Similar to last year, developers remain split on whether they trust AI output: 43% feel good about AI accuracy and 31% are skeptical. Developers learning to code are trusting AI accuracy more than their professional counterparts (49% vs. 42%).

How much do you trust the accuracy of the output from AI tools as part of your development workflow?



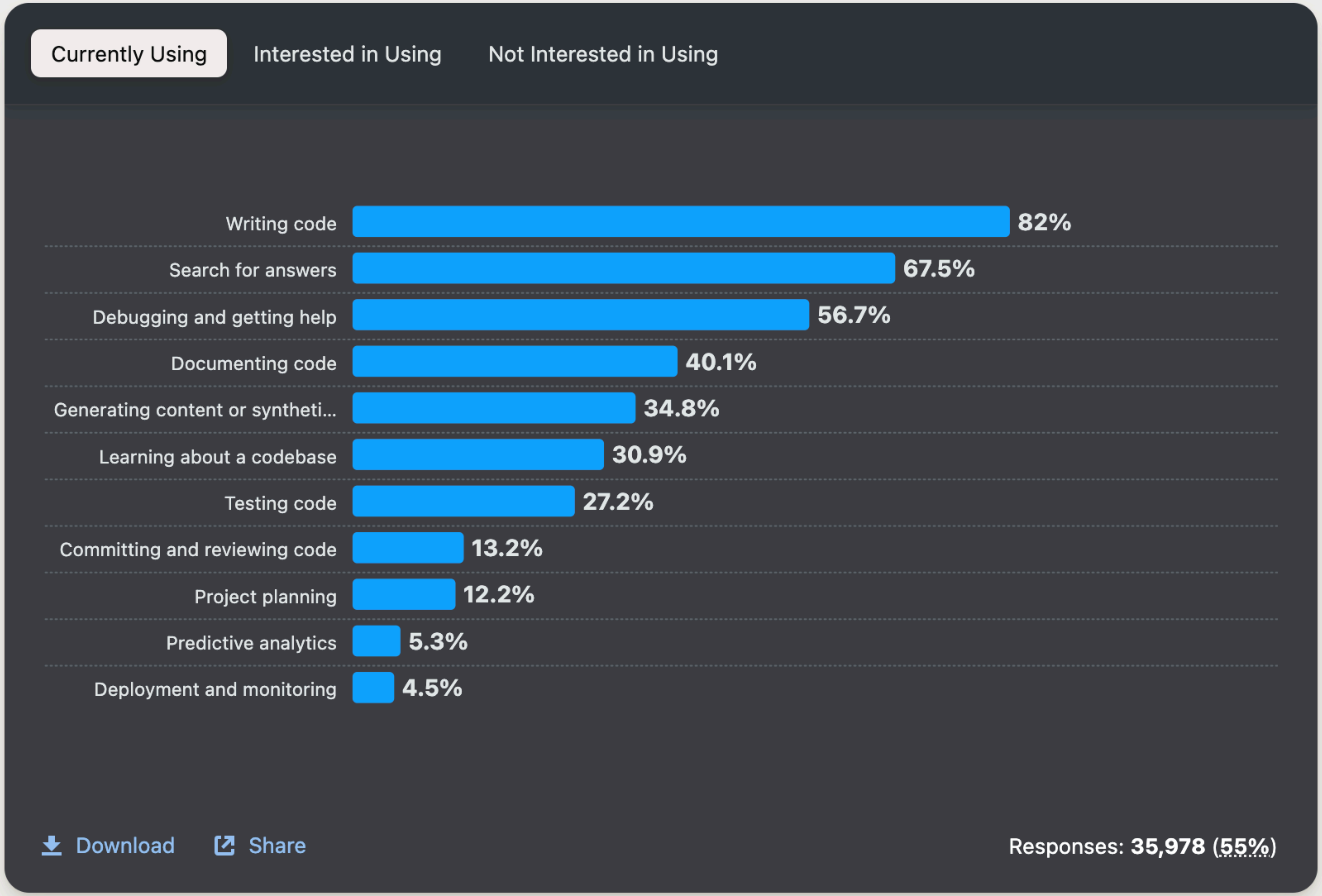
https://survey.stackoverflow.co/2024/ai

stackoverflow | 2024 Developer Survey

AI in the development workflow

Developers currently using AI tools mostly use them to write code (82%) and those who are interested but not yet using AI tools are mostly curious about testing code (46%). Developers with experience can trust AI tools to help write code to get started but perhaps know testing is a complex step best left to traditional processes.

Which parts of your development workflow are you currently using AI tools for and which are you interested in using AI tools for over the next year? Please select all that apply.



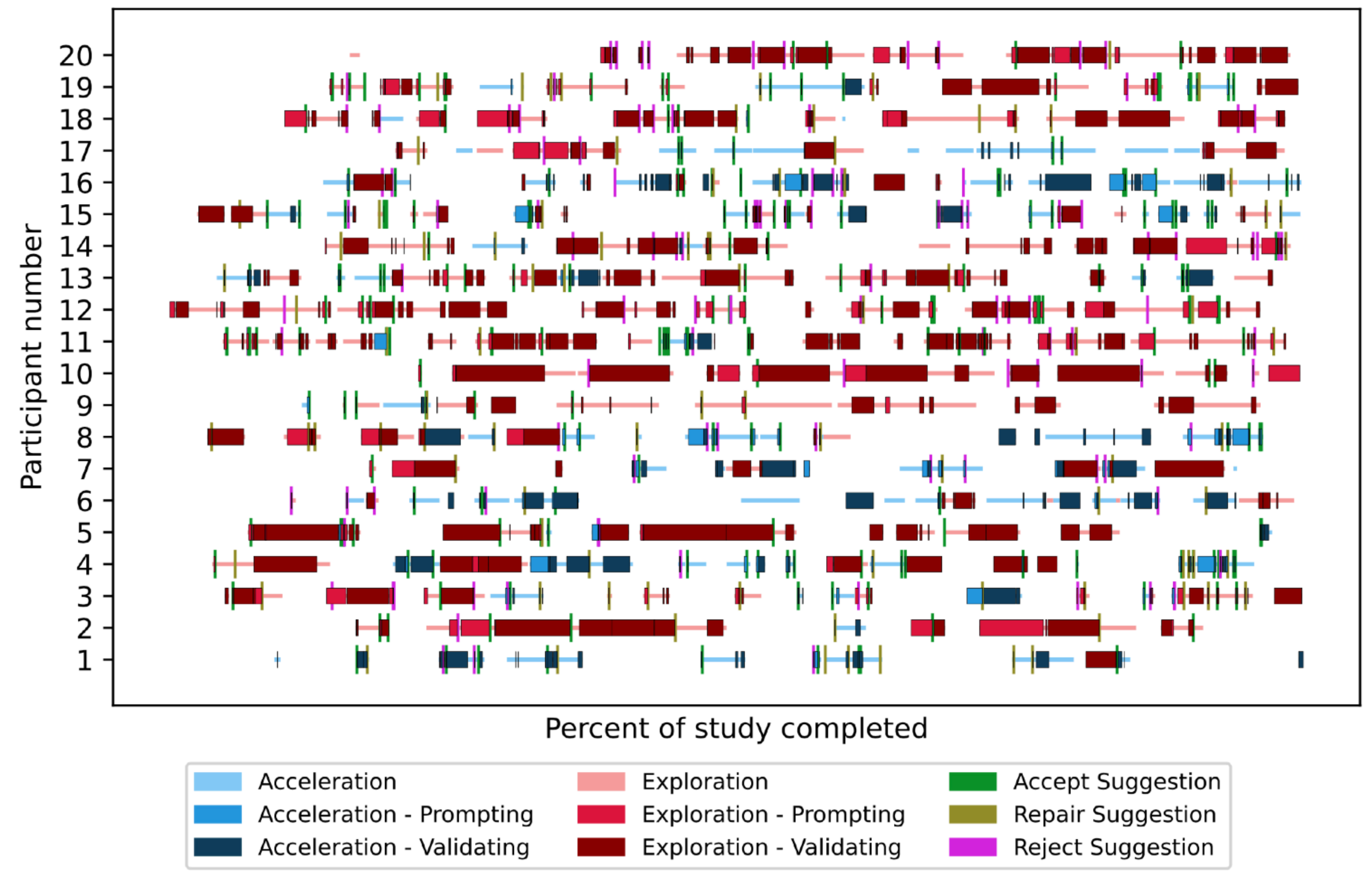
Task	Currently Using
Writing code	82%
Search for answers	67.5%
Debugging and getting help	56.7%
Documenting code	40.1%
Generating content or syntheti...	34.8%
Learning about a codebase	30.9%
Testing code	27.2%
Committing and reviewing code	13.2%
Project planning	12.2%
Predictive analytics	5.3%
Deployment and monitoring	4.5%

Responses: **35,978** (55%)

https://survey.stackoverflow.co/2024/ai

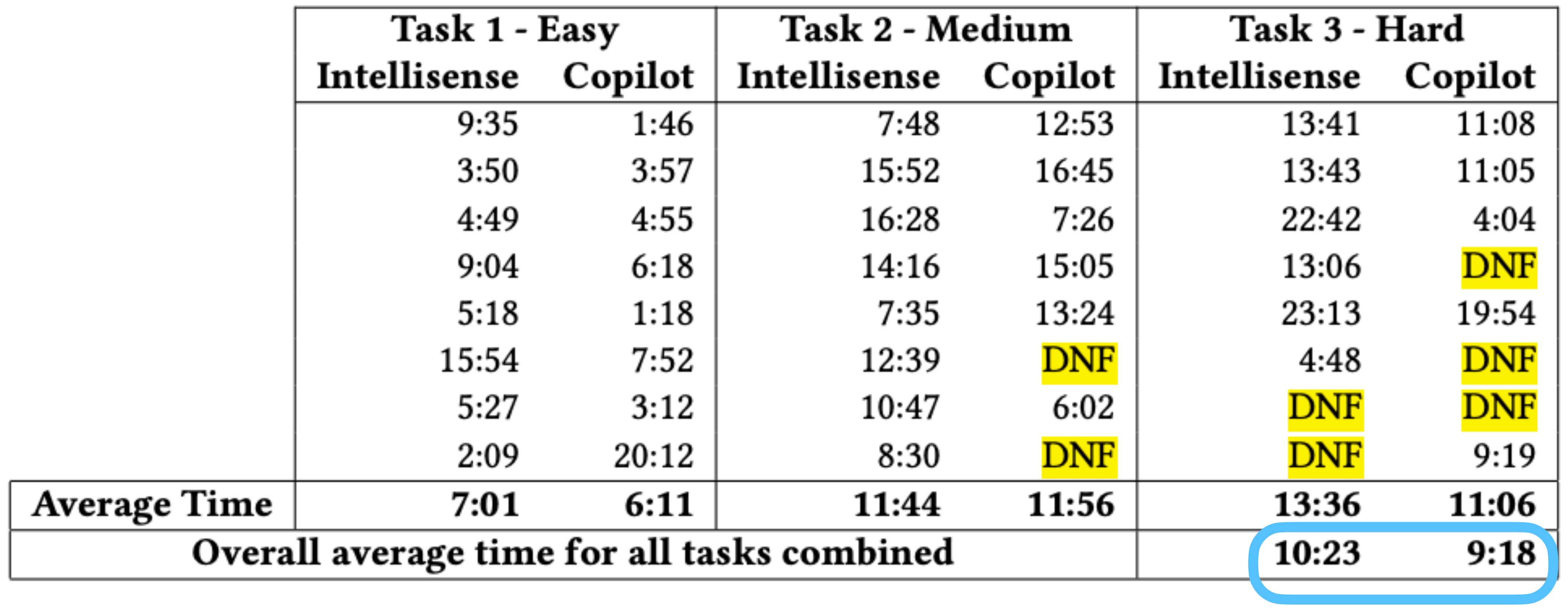
Grounded CoPilot

- Devs use code completion LLMs in two modes
- **Acceleration**: completing thought process
  - Developers formulate detailed idea for code
  - Long suggestions break flow
  - Skim suggestions to find one that matches expectations
- **Exploration**: novel tasks & unexpected behavior
  - Generate many solutions, mix and match solutions
  - Carefully validate by testing & reading docs



Evaluating the Usability of Code Generation Tools Powered by LLMs

- Developers felt more productive, but not significantly faster
- Replaced StackOverflow for API interactions, but only one suggestion & solutions often had defects
- Debugging sometimes harder without knowledge of how the code should work
- Sometimes suggested approaches that led participants in the direction of bad solutions

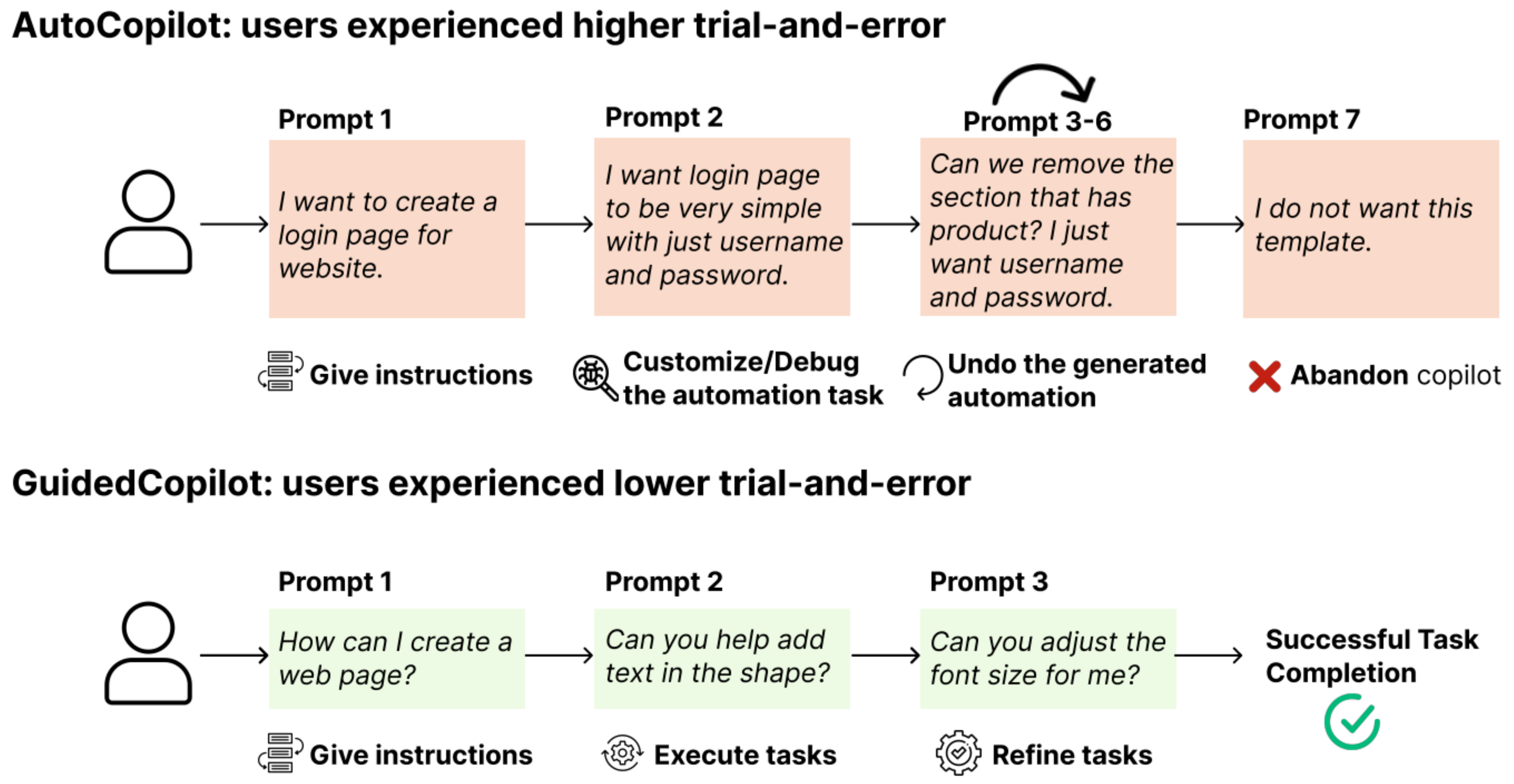


	Task 1 - Easy		Task 2 - Medium		Task 3 - Hard	
	Intellisense	**Copilot**	**Intellisense**	**Copilot**	**Intellisense**	**Copilot**
	9:35	1:46	7:48	12:53	13:41	11:08
	3:50	3:57	15:52	16:45	13:43	11:05
	4:49	4:55	16:28	7:26	22:42	4:04
	9:04	6:18	14:16	15:05	13:06	DNF
	5:18	1:18	7:35	13:24	23:13	19:54
	15:54	7:52	12:39	DNF	4:48	DNF
	5:27	3:12	10:47	6:02	DNF	DNF
	2:09	20:12	8:30	DNF	DNF	9:19
Average Time	**7:01**	**6:11**	**11:44**	**11:56**	**13:36**	**11:06**
Overall average time for all tasks combined					**10:23**	**9:18**

not significant (p = 0.53)

Perceptions of Paradigms of Automation

- For complex tasks, developers value guiding LLM rather than full automation


AutoCopilot: users experienced higher trial-and-error
Prompt 1
I want to create a login page for website.
Prompt 2
I want login page to be very simple with just username and password.
Prompt 3-6
Can we remove the section that has product? I just want username and password.
Prompt 7
I do not want this template.
Give instructions
Customize/Debug the automation task
Undo the generated automation
Abandon copilot
GuidedCopilot: users experienced lower trial-and-error
Prompt 1
How can I create a web page?
Prompt 2
Can you help add text in the shape?
Prompt 3
Can you adjust the font size for me?
Successful Task Completion
Give instructions
Execute tasks
Refine tasks

- How can developers understand the code LLMs generate?
- How much understanding is still necessary?
- How do developers figure out what's wrong when it doesn't work?

Dagstuhl Seminar 22231

Theories of Programming

(Jun 06 – Jun 10, 2022)

- Amy Ko (University of Washington - Seattle, US)
- Thomas D. LaToza (George Mason University - Fairfax, US)
- David C. Shepherd (Virginia Commonwealth University - Richmond, US)
- Dag Sjøberg (University of Oslo, NO)

Mature scientific disciplines are characterized by their theories, synthesizing what is known about phenomena into forms which generate falsifiable predictions about the world. In computer science, the role of synthesizing ideas has largely been through formalisms that describe how programs compute. However, just as important are scientific theories about how programmers write these programs. For example, software engineering research has increasingly begun gathering data, through observations, surveys, interviews, and analysis of artifacts, about the nature of programming work and the challenges developers face, and evaluating novel programming tools through controlled experiments with software developers. Computer science education and human-computer interaction research has done similar work, but for people with different levels of experience and ages learning to write programs. But data from such empirical studies is often left isolated, rather than combined into useful theories which explain all of the empirical results. This lack of theory makes it harder to predict in which contexts programming languages, tools, and pedagogy will actually help people successfully write and learn to create software.

Computer science needs scientific theories that synthesize what we believe to be true about programming and offer falsifiable predictions. Whether or not a theory is ultimately found to be consistent with evidence or discarded, theories offer a clear statement about our current understanding, helping us in prioritizing studies, generalizing study results from individual empirical results to more general understanding of phenomena, and offering the ability to design tools in ways that are consistent with current knowledge.

Theories of Program Comprehension in the Age of LLMs

- How can developers still understand the code being generated?

  ⇒ Theories of Information Needs in Programming

- To what extent do developers really have to understand the code being written?

  ⇒ Theories of Information Hiding

- How do developers figure out what's wrong when it doesn't work?

  ⇒ Theories of Debugging

Theories of Program Comprehension in the Age of LLMs

- **How can developers still understand the code being generated?**

  ⇒ **Theories of Information Needs in Programming**

- To what extent do developers really have to understand the code being written?

  ⇒ Theories of Information Hiding

- How do developers figure out what's wrong when it doesn't work?

  ⇒ Theories of Debugging

Theories of Information Needs in Programming

- Developers ask **questions**

What does this do when input is null?

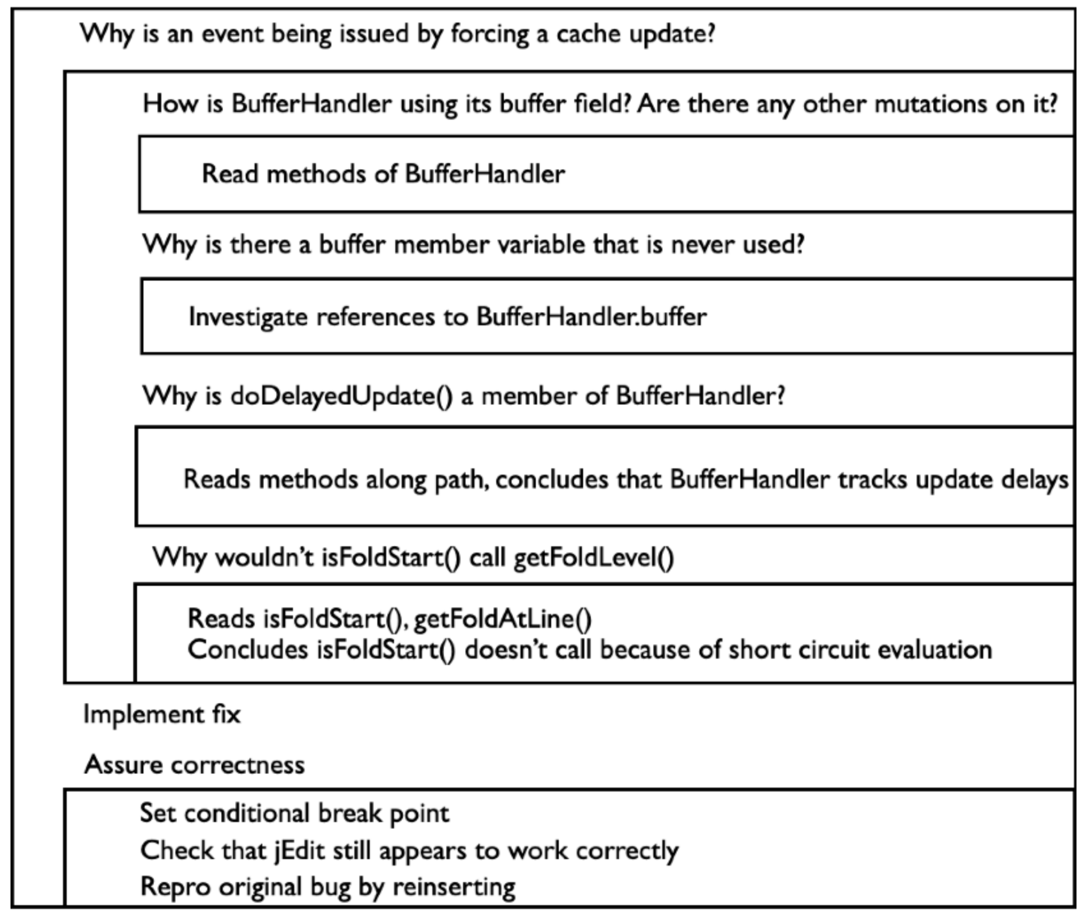
What part of this is being done client side and what part server side?

- Questions are **task-specific**

debugging refactoring testing testing

- Answering questions raises **more questions.**

- Tool which successfully supports the questions a developer asks increases their productivity



Why is an event being issued by forcing a cache update?

How is BufferHandler using its buffer field? Are there any other mutations on it?

Read methods of BufferHandler

Why is there a buffer member variable that is never used?

Investigate references to BufferHandler.buffer

Why is doDelayedUpdate() a member of BufferHandler?

Reads methods along path, concludes that BufferHandler tracks update delays

Why wouldn't isFoldStart() call getFoldLevel()

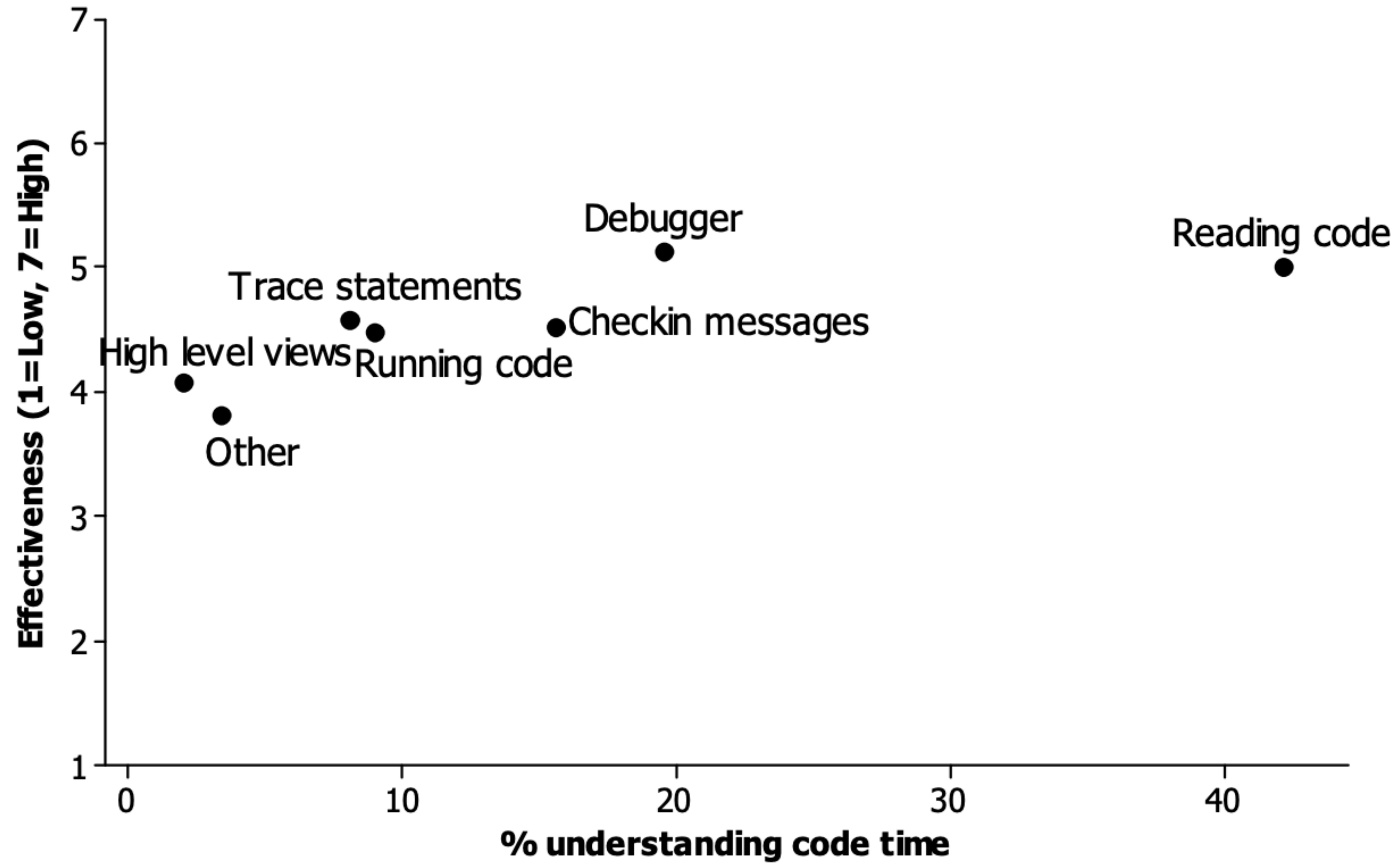
Reads isFoldStart(), getFoldAtLine()
Concludes isFoldStart() doesn't call because of short circuit evaluation

Implement fix

Assure correctness

Set conditional break point
Check that jEdit still appears to work correctly
Repro original bug by reinserting

Techniques for understanding code



Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In Proceedings of the 28th international conference on Software engineering (ICSE '06), Experience Report, 492–501. https://doi.org/10.1145/1134285.1134355

What makes understanding code hard?

- Questions developers ask about code that are hard to answer.
- May require substantial time and effort to answer.
- May lead to many other questions to answer

Longest investigation activities

	Time (mins)
How is this data structure being mutated in this code?	83
"Where [is] the code assuming that the tables are already there?"	53
How [does] application state change when *m* is called denoting startup completion?	50
What decisions might be incompatible with reuse in new context?	24
"Is [there] another reason why *status* could be non-zero?"	11

Thomas D. LaToza and Brad A. Myers. 2010. Developers ask reachability questions. In International Conference on Software Engineering, 185–194. https://doi.org/10.1145/1806799.1806829

Rationale (42)

Why was it done this way? (14) [15][7]
Why wasn't it done this other way? (15)
Was this intentional, accidental, or a hack? (9)[15]
How did this ever work? (4)

Debugging (26)

How did this runtime state occur? (12) [15]
What runtime state changed when this executed? (2)
Where was this variable last changed? (1)
How is this object different from that object? (1)
Why didn't this happen? (3)
How do I debug this bug in this environment? (3)
In what circumstances does this bug occur? (3) [15]
Which team's component caused this bug? (1)

Intent and Implementation (32)

What is the intent of this code? (12) [15]
What does this do (6) in this case (10)? (16) [24]
How does it implement this behavior? (4) [24]

Refactoring (25)

Is there functionality or code that could be refactored? (4)
Is the existing design a good design? (2)
Is it possible to refactor this? (9)
How can I refactor this (2) without breaking existing users(7)? (9)
Should I refactor this? (1)
Are the benefits of this refactoring worth the time investment? (3)

History (23)

When, how, by whom, and why was this code changed or inserted? (13)[7]
What else changed when this code was changed or inserted? (2)
How has it changed over time? (4)[7]
Has this code always been this way? (2)
What recent changes have been made? (1)[15][7]
Have changes in another branch been integrated into this branch? (1)

Implications (21)

What are the implications of this change for (5) API clients (5), security (3), concurrency (3), performance (2), platforms (1), tests (1), or obfuscation (1)? (21) [15][24]

Testing (20)

Is this code correct? (6) [15]
How can I test this code or functionality? (9)
Is this tested? (3)
Is the test or code responsible for this test failure? (1)
Is the documentation wrong, or is the code wrong? (1)

Implementing (19)

How do I implement this (8), given this constraint (2)? (10)
Which function or object should I pick? (2)
What's the best design for implementing this? (7)

Control flow (19)

In what situations or user scenarios is this called? (3) [15][24]
What parameter values does each situation pass to this method? (1)
What parameter values could lead to this case? (1)
What are the possible actual methods called by dynamic dispatch here? (6)
How do calls flow across process boundaries? (1)
How many recursive calls happen during this operation? (1)
Is this method or code path called frequently, or is it dead? (4)
What throws this exception? (1)
What is catching this exception? (1)

Contracts (17)

What assumptions about preconditions does this code make? (5)
What assumptions about pre(3)/post(2)conditions can be made?
What exceptions or errors can this method generate? (2)
What are the constraints on or normal values of this variable? (2)
What is the correct order for calling these methods or initializing these objects? (2)
What is responsible for updating this field? (1)

Performance (16)

What is the performance of this code (5) on a large, real dataset (3)? (8)
Which part of this code takes the most time? (4)
Can this method have high stack consumption from recursion? (1)
How big is this in memory? (2)
How many of these objects get created? (1)

Teammates (16)

Who is the owner or expert for this code? (3)[7]
How do I convince my teammates to do this the "right way"? (12)
Did my teammates do this? (1)

Policies (15)

What is the policy for doing this? (10) [24]
Is this the correct policy for doing this? (2) [15]
How is the allocation lifetime of this object maintained? (3)

Type relationships (15)

What are the composition, ownership, or usage relationships of this type? (5) [24]
What is this type's type hierarchy? (4) [24]
What implements this interface? (4) [24]
Where is this method overridden? (2)

Data flow (14)

What is the original source of this data? (2) [15]
What code directly or indirectly uses this data? (5)
Where is the data referenced by this variable modified? (2)
Where can this global variable be changed? (1)
Where is this data structure used (1) for this purpose (1)? (2) [24]
What parts of this data structure are modified by this code? (1) [24]
What resources is this code using? (1)

Location (13)

Where is this functionality implemented? (5) [24]
Is this functionality already implemented? (5) [15]
Where is this defined? (3)

Building and branching (11)

Should I branch or code against the main branch? (1)
How can I move this code to this branch? (1)
What do I need to include to build this? (3)
What includes are unnecessary? (2)
How do I build this without doing a full build? (1)
Why did the build break? (2)[59]
Which preprocessor definitions were active when this was built? (1)

Architecture (11)
How does this code interact with libraries? (4)
What is the architecture of the code base? (3)
How is this functionality organized into layers? (1)
Is our API understandable and flexible? (3)

Concurrency (9)

What threads reach this code (4) or data structure (2)? (6)
Is this class or method thread-safe? (2)
What members of this class does this lock protect? (1)

Dependencies (5)

What depends on this code or design decision? (4)[7]
What does this code depend on? (1)

Method properties (2)
How big is this code? (1)
How overloaded are the parameters to this function? (1)

What does this do?
What does these functions do?

What does this do in this case?
What happens if an exception is thrown?
What happens if this operation times out?
What happens if the remote service is slow?

What is the intent of the code?
What is it trying to accomplish?

How does it implement this behavior?
How is this data aggregated and how is it translated from one place to another.
How does this class (or collection of classes) fulfill the functional feature of the application?

Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer questions about code. In Evaluation and Usability of Programming Languages and Tools (PLATEAU '10), 1–6. https://doi.org/10.1145/1937117.1937125

What depends on this code or design decision?
What else depends on this code?
Who else uses this code / function. (i.e. If we change this, what will break simply because someone else has found a way to use this and we don't even know they are doing so...)

What are the implications of this change for API clients, security, concurrency, performance, platforms, tests, or obfuscation?
What is the implication of these changes in terms of the backward compatibility?
Across components with a code base the size of complete applications, what are the implications of a functional change in base storage to all accessors in the system (including clients of applications built on top of the place where the change is occurring)

How can I refactor this without breaking existing users?
How can I refactor this piece w/o causing an avalanche of new places to refactor?

Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer questions about code. In Evaluation and Usability of Programming Languages and Tools (PLATEAU '10), 1–6. https://doi.org/10.1145/1937117.1937125

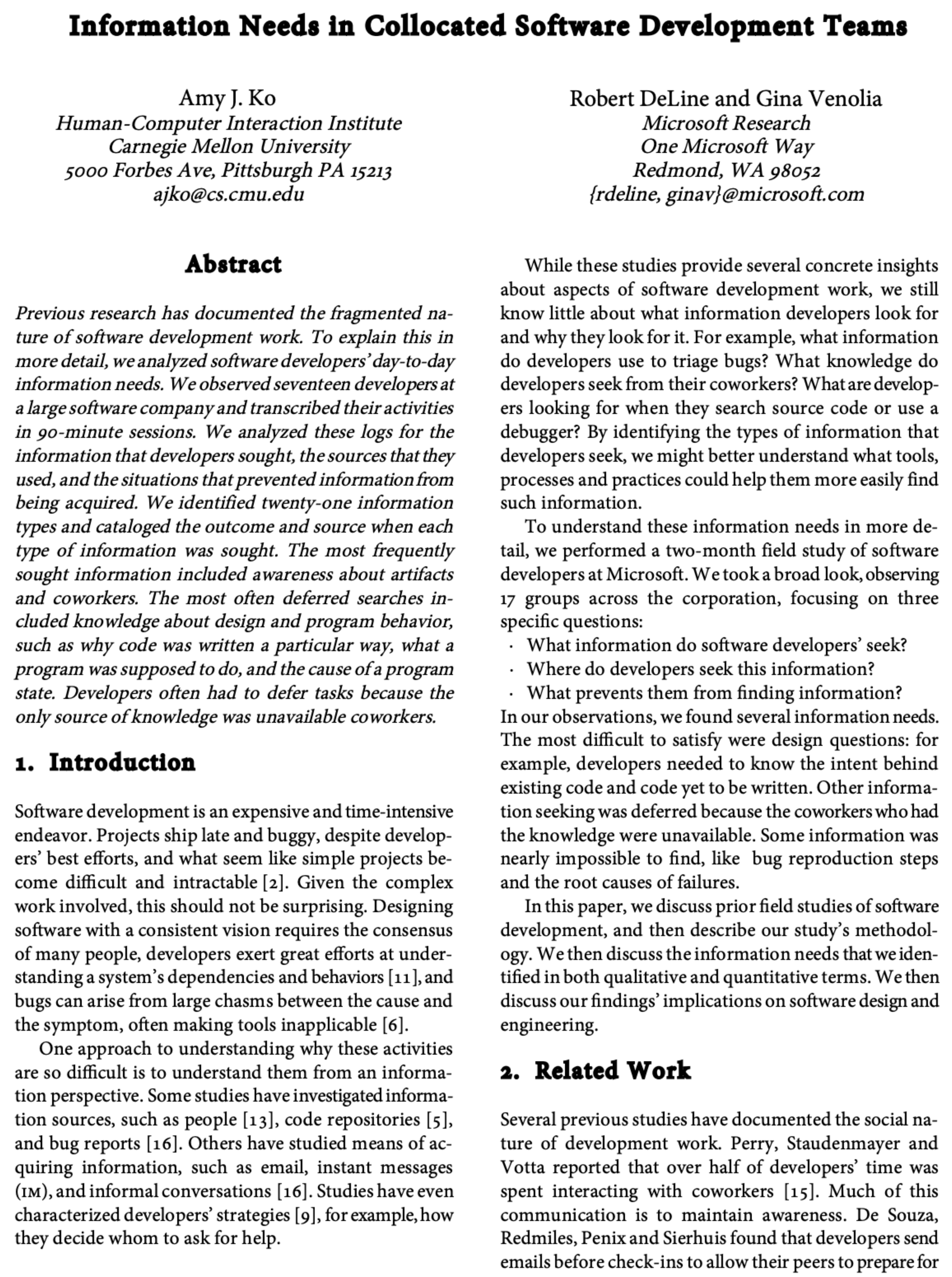
Why was it done this way?
Why was this code structured in this way?
Why was this done this way? Is there some reason for this ancient code doing what it does that I'm missing?

Why wasn't it done this other way?
Why didn't they use this method/object/interface as it appears to have been designed?
Why did the original developer not use library function X? (was there a good reason or just ignorance)

Was this intentional, accidental, or a hack?
Is the lack of parameter validation (most often lack of null checks) intentional or incidental?
Is the lack of ''sealed'' on the class intentional or incidental? If intentional, why? (assuming no virtual methods are present).

Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer questions about code. In Evaluation and Usability of Programming Languages and Tools (PLATEAU '10), 1–6. https://doi.org/10.1145/1937117.1937125

Studies of questions developers ask



Information Needs in Collocated Software Development Teams

Amy J. Ko
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213
ajko@cs.cmu.edu

Robert DeLine and Gina Venolia
Microsoft Research
One Microsoft Way
Redmond, WA 98052
{rdeline, ginav}@microsoft.com

Abstract

Previous research has documented the fragmented nature of software development work. To explain this in more detail, we analyzed software developers' day-to-day information needs. We observed seventeen developers at a large software company and transcribed their activities in 90-minute sessions. We analyzed these logs for the information that developers sought, the sources that they used, and the situations that prevented information from being acquired. We identified twenty-one information types and cataloged the outcome and source when each type of information was sought. The most frequently sought information included awareness about artifacts and coworkers. The most often deferred searches included knowledge about design and program behavior, such as why code was written a particular way, what a program was supposed to do, and the cause of a program state. Developers often had to defer tasks because the only source of knowledge was unavailable coworkers.

1. Introduction

Software development is an expensive and time-intensive endeavor. Projects ship late and buggy, despite developers' best efforts, and what seem like simple projects become difficult and intractable [2]. Given the complex work involved, this should not be surprising. Designing software with a consistent vision requires the consensus of many people, developers exert great efforts at understanding a system's dependencies and behaviors [11], and bugs can arise from large chasms between the cause and the symptom, often making tools inapplicable [6].

One approach to understanding why these activities are so difficult is to understand them from an information perspective. Some studies have investigated information sources, such as people [13], code repositories [5], and bug reports [16]. Others have studied means of acquiring information, such as email, instant messages (IM), and informal conversations [16]. Studies have even characterized developers' strategies [9], for example, how they decide whom to ask for help.

While these studies provide several concrete insights about aspects of software development work, we still know little about what information developers look for and why they look for it. For example, what information do developers use to triage bugs? What knowledge do developers seek from their coworkers? What are developers looking for when they search source code or use a debugger? By identifying the types of information that developers seek, we might better understand what tools, processes and practices could help them more easily find such information.

To understand these information needs in more detail, we performed a two-month field study of software developers at Microsoft. We took a broad look, observing 17 groups across the corporation, focusing on three specific questions:

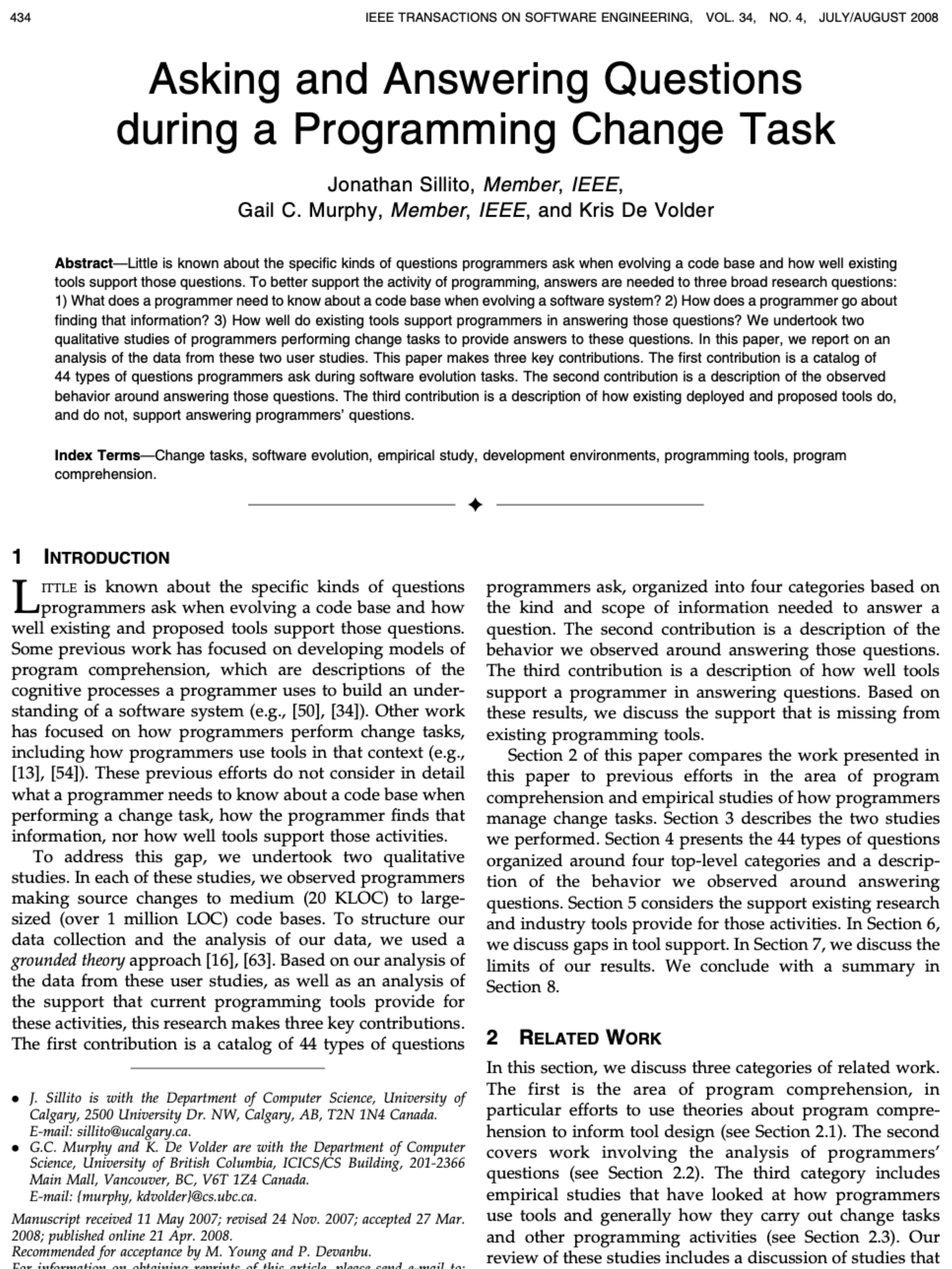
- What information do software developers' seek?
- Where do developers seek this information?
- What prevents them from finding information?

In our observations, we found several information needs. The most difficult to satisfy were design questions: for example, developers needed to know the intent behind existing code and code yet to be written. Other information seeking was deferred because the coworkers who had the knowledge were unavailable. Some information was nearly impossible to find, like bug reproduction steps and the root causes of failures.

In this paper, we discuss prior field studies of software development, and then describe our study's methodology. We then discuss the information needs that we identified in both qualitative and quantitative terms. We then discuss our findings' implications on software design and engineering.

2. Related Work

Several previous studies have documented the social nature of development work. Perry, Staudenmayer and Votta reported that over half of developers' time was spent interacting with coworkers [15]. Much of this communication is to maintain awareness. De Souza, Redmiles, Penix and Sierhuis found that developers send emails before check-ins to allow their peers to prepare for



Asking and Answering Questions during a Programming Change Task

Jonathan Sillito, *Member, IEEE*,
Gail C. Murphy, *Member, IEEE*, and Kris De Volder

Abstract—Little is known about the specific kinds of questions programmers ask when evolving a code base and how well existing tools support those questions. To better support the activity of programming, answers are needed to three broad research questions: 1) What does a programmer need to know about a code base when evolving a software system? 2) How does a programmer go about finding that information? 3) How well do existing tools support programmers in answering those questions? We undertook two qualitative studies of programmers performing change tasks to provide answers to these questions. In this paper, we report on an analysis of the data from these two user studies. This paper makes three key contributions. The first contribution is a catalog of 44 types of questions programmers ask during software evolution tasks. The second contribution is a description of the observed behavior around answering those questions. The third contribution is a description of how existing deployed and proposed tools do, and do not, support answering programmers' questions.

Index Terms—Change tasks, software evolution, empirical study, development environments, programming tools, program comprehension.

1 INTRODUCTION

LITTLE is known about the specific kinds of questions programmers ask when evolving a code base and how well existing and proposed tools support those questions. Some previous work has focused on developing models of program comprehension, which are descriptions of the cognitive processes a programmer uses to build an understanding of a software system (e.g., [50], [34]). Other work has focused on how programmers perform change tasks, including how programmers use tools in that context (e.g., [13], [54]). These previous efforts do not consider in detail what a programmer needs to know about a code base when performing a change task, how the programmer finds that information, nor how well tools support those activities.

To address this gap, we undertook two qualitative studies. In each of these studies, we observed programmers making source changes to medium (20 KLOC) to large-sized (over 1 million LOC) code bases. To structure our data collection and the analysis of our data, we used a *grounded theory* approach [16], [63]. Based on our analysis of the data from these user studies, as well as an analysis of the support that current programming tools provide for these activities, this research makes three key contributions. The first contribution is a catalog of 44 types of questions programmers ask, organized into four categories based on the kind and scope of information needed to answer a question. The second contribution is a description of the behavior we observed around answering those questions. The third contribution is a description of how well tools support a programmer in answering questions. Based on these results, we discuss the support that is missing from existing programming tools.

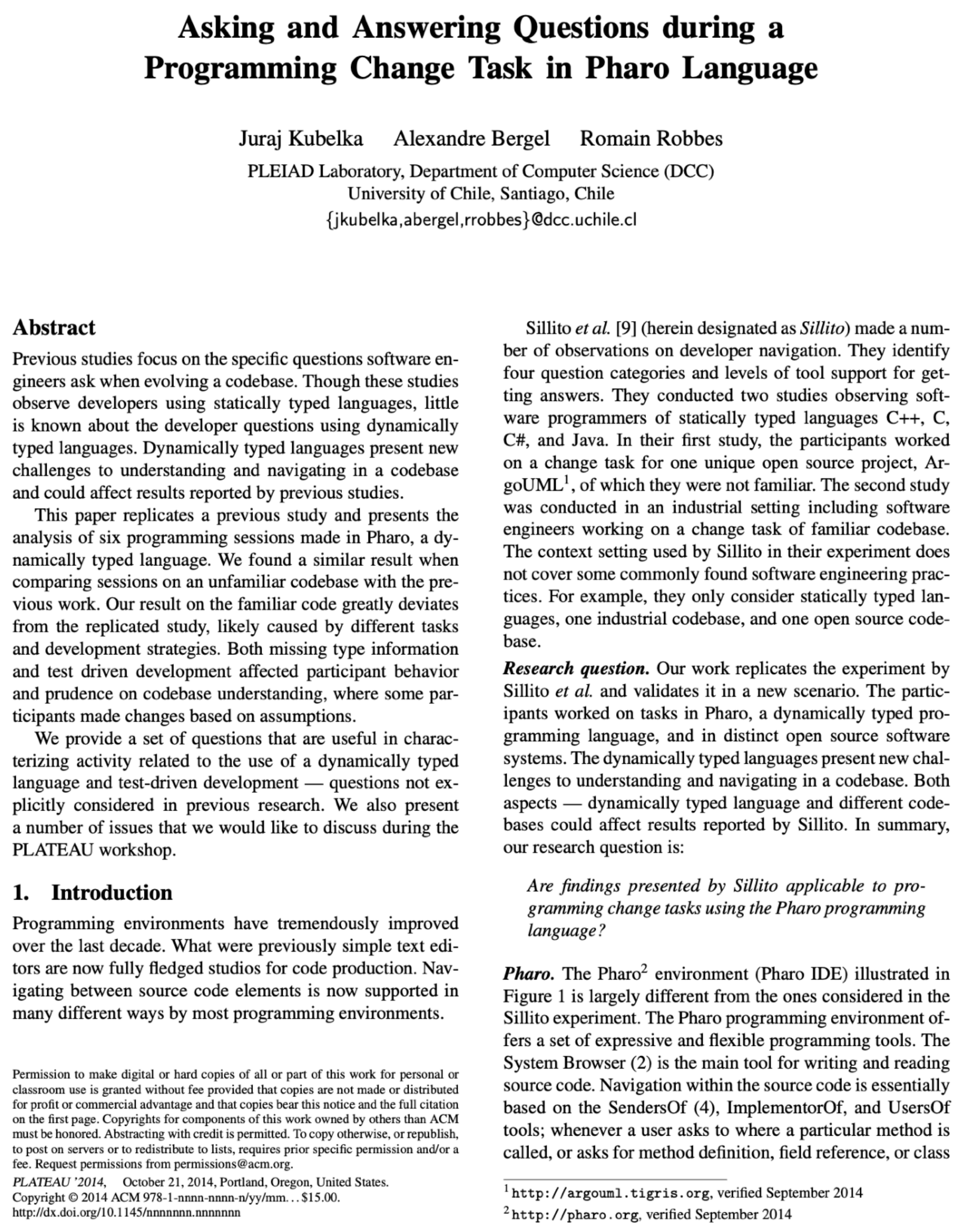
Section 2 of this paper compares the work presented in this paper to previous efforts in the area of program comprehension and empirical studies of how programmers manage change tasks. Section 3 describes the two studies we performed. Section 4 presents the 44 types of questions organized around four top-level categories and a description of the behavior we observed around answering questions. Section 5 considers the support existing research and industry tools provide for those activities. In Section 6, we discuss gaps in tool support. In Section 7, we discuss the limits of our results. We conclude with a summary in Section 8.

2 RELATED WORK

In this section, we discuss three categories of related work. The first is the area of program comprehension, in particular efforts to use theories about program comprehension to inform tool design (see Section 2.1). The second covers work involving the analysis of programmers' questions (see Section 2.2). The third category includes empirical studies that have looked at how programmers use tools and generally how they carry out change tasks and other programming activities (see Section 2.3). Our review of these studies includes a discussion of studies that

- *J. Sillito is with the Department of Computer Science, University of Calgary, 2500 University Dr. NW, Calgary, AB, T2N 1N4 Canada. E-mail: sillito@ucalgary.ca.*
- *G.C. Murphy and K. De Volder are with the Department of Computer Science, University of British Columbia, ICICS/CS Building, 201-2366 Main Mall, Vancouver, BC, V6T 1Z4 Canada. E-mail: {murphy, kdvolder}@cs.ubc.ca.*

Manuscript received 11 May 2007; revised 24 Nov. 2007; accepted 27 Mar. 2008; published online 21 Apr. 2008.
Recommended for acceptance by M. Young and P. Devanbu.
For information on obtaining reprints of this article, please send e-mail to:



Asking and Answering Questions during a Programming Change Task in Pharo Language

Juraj Kubelka Alexandre Bergel Romain Robbes
PLEIAD Laboratory, Department of Computer Science (DCC)
University of Chile, Santiago, Chile
{jkubelka,abergel,rrobbes}@dcc.uchile.cl

Abstract

Previous studies focus on the specific questions software engineers ask when evolving a codebase. Though these studies observe developers using statically typed languages, little is known about the developer questions using dynamically typed languages. Dynamically typed languages present new challenges to understanding and navigating in a codebase and could affect results reported by previous studies.

This paper replicates a previous study and presents the analysis of six programming sessions made in Pharo, a dynamically typed language. We found a similar result when comparing sessions on an unfamiliar codebase with the previous work. Our result on the familiar code greatly deviates from the replicated study, likely caused by different tasks and development strategies. Both missing type information and test driven development affected participant behavior and prudence on codebase understanding, where some participants made changes based on assumptions.

We provide a set of questions that are useful in characterizing activity related to the use of a dynamically typed language and test-driven development — questions not explicitly considered in previous research. We also present a number of issues that we would like to discuss during the PLATEAU workshop.

1. Introduction

Programming environments have tremendously improved over the last decade. What were previously simple text editors are now fully fledged studios for code production. Navigating between source code elements is now supported in many different ways by most programming environments.

PLATEAU '2014, October 21, 2014, Portland, Oregon, United States.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Sillito *et al.* [9] (herein designated as *Sillito*) made a number of observations on developer navigation. They identify four question categories and levels of tool support for getting answers. They conducted two studies observing software programmers of statically typed languages C++, C, C#, and Java. In their first study, the participants worked on a change task for one unique open source project, ArgoUML[1], of which they were not familiar. The second study was conducted in an industrial setting including software engineers working on a change task of familiar codebase. The context setting used by Sillito in their experiment does not cover some commonly found software engineering practices. For example, they only consider statically typed languages, one industrial codebase, and one open source codebase.

Research question. Our work replicates the experiment by Sillito *et al.* and validates it in a new scenario. The participants worked on tasks in Pharo, a dynamically typed programming language, and in distinct open source software systems. The dynamically typed languages present new challenges to understanding and navigating in a codebase. Both aspects — dynamically typed language and different codebases could affect results reported by Sillito. In summary, our research question is:

> *Are findings presented by Sillito applicable to programming change tasks using the Pharo programming language?*

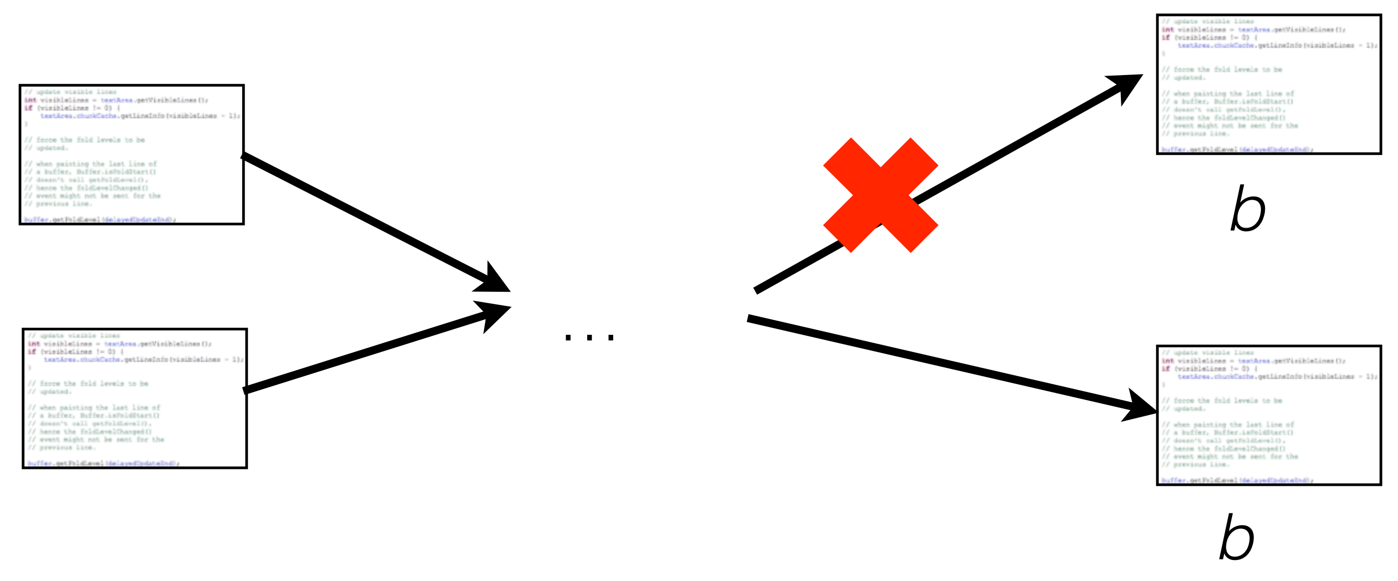
Pharo. The Pharo[2] environment (Pharo IDE) illustrated in Figure 1 is largely different from the ones considered in the Sillito experiment. The Pharo programming environment offers a set of expressive and flexible programming tools. The System Browser (2) is the main tool for writing and reading source code. Navigation within the source code is essentially based on the SendersOf (4), ImplementorOf, and UsersOf tools; whenever a user asks to where a particular method is called, or asks for method definition, field reference, or class

[1] `http://argouml.tigris.org`, verified September 2014
[2] `http://pharo.org`, verified September 2014

Failure in information needs

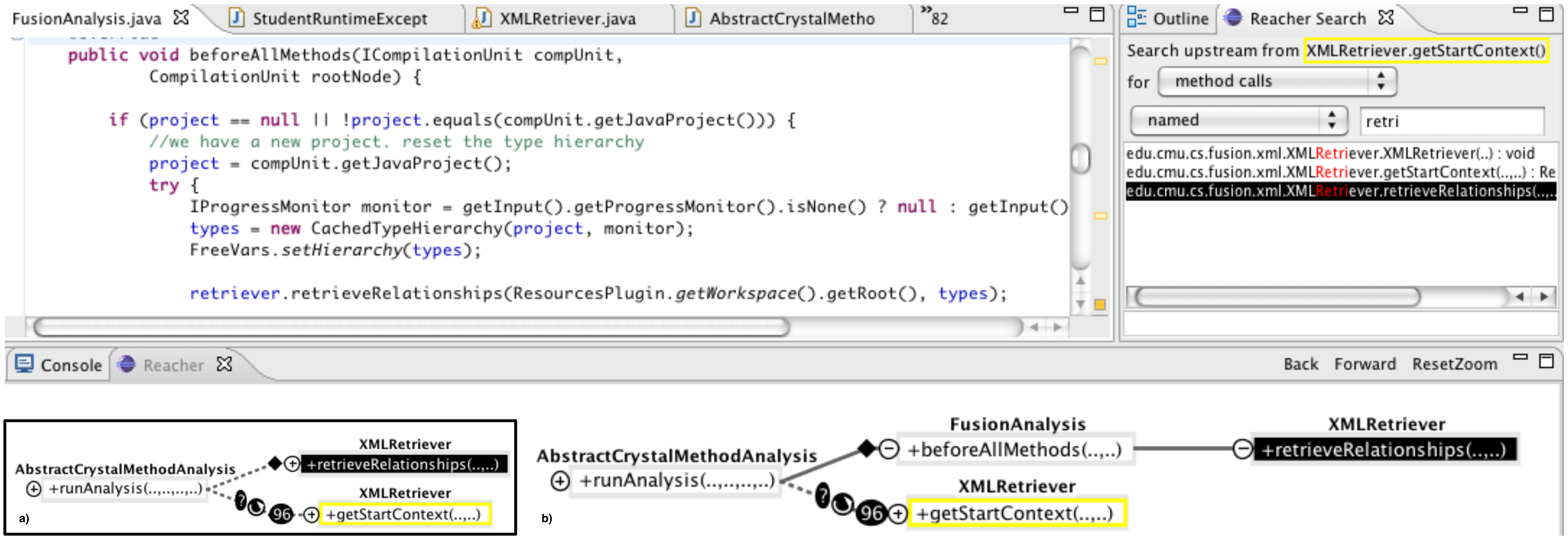
- Developers guess and make assumptions about answers to questions, and sometimes are wrong, leading to defects.

False belief held by developer	**Correct fact about control flow**
Method *a* need not call method *b*, as all calls to be are redundant.	*m* is called in several additional situations in which *n* has not been called.



Thomas D. LaToza and Brad A. Myers. 2010. Developers ask reachability questions. In International Conference on Software Engineering, 185–194. https://doi.org/10.1145/1806799.1806829

Supporting question answering with tools



FusionAnalysis.java | StudentRuntimeExcept | XMLRetriever.java | AbstractCrystalMetho | »82

```java
    public void beforeAllMethods(ICompilationUnit compUnit,
            CompilationUnit rootNode) {

        if (project == null || !project.equals(compUnit.getJavaProject())) {
            //we have a new project. reset the type hierarchy
            project = compUnit.getJavaProject();
            try {
                IProgressMonitor monitor = getInput().getProgressMonitor().isNone() ? null : getInput()
                types = new CachedTypeHierarchy(project, monitor);
                FreeVars.setHierarchy(types);

                retriever.retrieveRelationships(ResourcesPlugin.getWorkspace().getRoot(), types);
```

Outline | Reacher Search

Search upstream from XMLRetriever.getStartContext()
for: method calls
named: retri

```
edu.cmu.cs.fusion.xml.XMLRetriever.XMLRetriever(..) : void
edu.cmu.cs.fusion.xml.XMLRetriever.getStartContext(...,..) : Re
edu.cmu.cs.fusion.xml.XMLRetriever.retrieveRelationships(...,..
```

Console | Reacher — Back Forward ResetZoom

a)
AbstractCrystalMethodAnalysis
+runAnalysis(..,..,..,..)
XMLRetriever
+retrieveRelationships(..,..)
XMLRetriever
96
+getStartContext(..,..)
b)
AbstractCrystalMethodAnalysis
+runAnalysis(..,..,..,..)
FusionAnalysis
+beforeAllMethods(..,..)
XMLRetriever
+retrieveRelationships(..,..)
XMLRetriever
96
+getStartContext(..,..)

T. D. LaToza and B. A. Myers, “Visualizing call graphs,” in Proc. Symp. Visual Languages and Human-Centric Computing (VL/HCC), 2011, pp. 117–124. doi: 10.1109/VLHCC.2011.6070388.

Programming Schemas

```
import { useState } from 'react';

function SearchableVideoList({ videos }) {
  const [searchText, setSearchText] = useState('');
  const foundVideos = filterVideos(videos, searchText);
  return (
    <>
      <SearchInput
        value={searchText}
        onChange={newText => setSearchText(newText)} />
      <VideoList
        videos={foundVideos}
        emptyHeading={`No matches for “${searchText}”`} />
    </>
  );
}
```

Knowledgeable developers see code differently

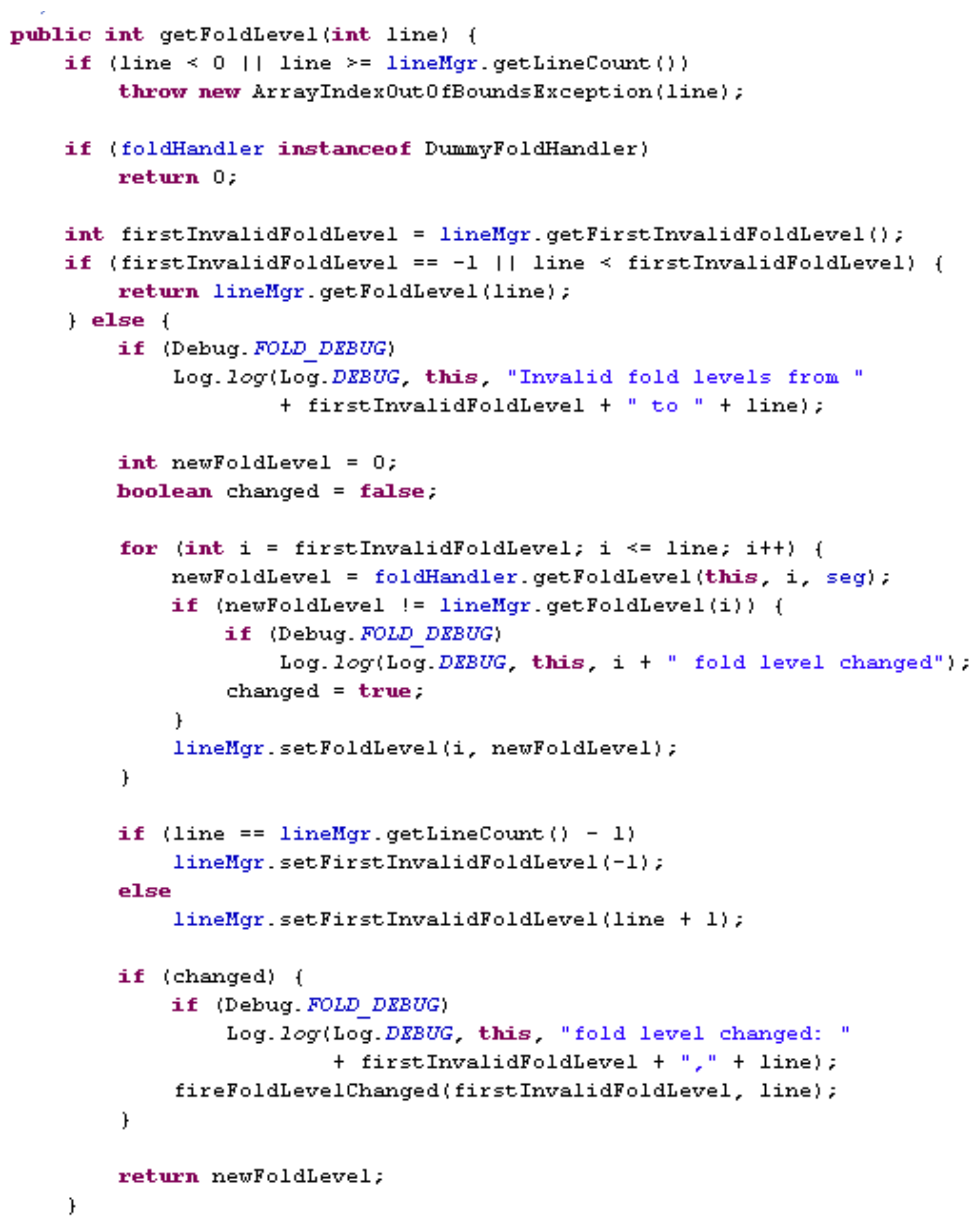
LESS EXPERIENCED DEVELOPERS

“What it did was it…computes the new line number and fires an event. But I didn’t see it change any state.” *(38 mins, 10 mins reading getFoldLevel)*

“So what it does, it starts off from this line, it has this firstInvalidFoldLevel, it goes through all these lines, it checks whether this fold information is correct or not, which is this newFoldLevel, this is supposed to be the correct fold level. If that is not the case in the data structure, it needs to change the state of the buffer. It creates this, it does this change, it sets the fold level of that line to the new fold level.” *(51 mins, 12 mins reading getFoldLevel)*

EXPERIENCED DEVELOPER

“Well, this is just updating a cache” *(1 min)*



```
public int getFoldLevel(int line) {
    if (line < 0 || line >= lineMgr.getLineCount())
        throw new ArrayIndexOutOfBoundsException(line);

    if (foldHandler instanceof DummyFoldHandler)
        return 0;

    int firstInvalidFoldLevel = lineMgr.getFirstInvalidFoldLevel();
    if (firstInvalidFoldLevel == -1 || line < firstInvalidFoldLevel) {
        return lineMgr.getFoldLevel(line);
    } else {
        if (Debug.FOLD_DEBUG)
            Log.log(Log.DEBUG, this, "Invalid fold levels from "
                    + firstInvalidFoldLevel + " to " + line);

        int newFoldLevel = 0;
        boolean changed = false;

        for (int i = firstInvalidFoldLevel; i <= line; i++) {
            newFoldLevel = foldHandler.getFoldLevel(this, i, seg);
            if (newFoldLevel != lineMgr.getFoldLevel(i)) {
                if (Debug.FOLD_DEBUG)
                    Log.log(Log.DEBUG, this, i + " fold level changed");
                changed = true;
            }
            lineMgr.setFoldLevel(i, newFoldLevel);
        }

        if (line == lineMgr.getLineCount() - 1)
            lineMgr.setFirstInvalidFoldLevel(-1);
        else
            lineMgr.setFirstInvalidFoldLevel(line + 1);

        if (changed) {
            if (Debug.FOLD_DEBUG)
                Log.log(Log.DEBUG, this, "fold level changed: "
                        + firstInvalidFoldLevel + "," + line);
            fireFoldLevelChanged(firstInvalidFoldLevel, line);
        }

        return newFoldLevel;
    }
```

51% CALLBACK IDIOMS

29% BIND TARGETS IDENTIFYING OR CHOOSING AN EVENT, LIFECYCLE HOOK, OR TRIGGER TO REGISTER A CALLBACK

CB1 *Unidentified Target:*
desired bind target → target name & code fragment

CB2 *Constrained Target:*
bind target code fragment → API rules making fragment (in)valid

CB3 *Confused Target:*
current & desired bind targets → API use differences, new target's code fragment

25% CALLBACK CONTEXTS IDENTIFYING WHEN THE CALLBACK IS DISPATCHED, USING ITS ARGUMENTS, OR OTHER RELATED OBJECTS

CB4 *Improper Scheduling:*
callback code fragments & desired schedule → correct callback order & code fix

CB5 *Unidentified State:*
desired state → API rationale for identifying state & code fragment to obtain it

CB6 *Missed Callbacks:*
callback code fragment → API rationale & state required for callback to occur

23% BIND CONFIGURATIONS SETTING OPTIONS OF A CALLBACK TRIGGER, OR MODIFYING PARAMETERS OF ITS BIND MECHANISM

CB7 *Incorrect Bind Parameters:*
callback parameter fragments & desired behavior → correct code fragments

CB8 *Misconfigured Framework:*
framework configuration fragments & desired behavior → correct framework code

42% GRAPHICAL IDIOMS

37% GRAPHICAL SETTERS UPDATING GRAPHICAL PROPERTIES OF THE LAYOUT VIA API (DOM ACCESS METHODS, CSS SELECTORS)

GB1 *Unidentified Setter:*
visual property change → code fragment to mutate property

GB2 *Unobservable Setter:*
setterA & visual property change → setterB to mutate property

GB3 *Indirect Setter:*
setterA → elements which inherit properties from setterA or occlude mutations

GB4 *Overwritten Setter:*
setterA → setterB overwriting setterA & code fragments with alternative fixes

21% GRAPHICAL QUERIES RETRIEVING GRAPHICAL ELEMENTS OR SIMILAR REPRESENTATIONS VIA API (DOM ACCESS METHODS, CSS SELECTORS)

GB5 *Incomplete Query:*
queryA and desired elements to be matched → queryB matching those elements

GB6 *Outdated Query:*
queryA → changes to query result set over time & code fragment fixing it

GB7 *Overwritten Query:*
queryA → queryB intersecting queryA's mutations & code fragment fixing queryA

8% GRAPHICAL GETTERS OBTAINING GRAPHICAL PROPERTIES OF THE LAYOUT VIA API METHODS

GB8 *Unidentified Getter:*
visual property → getter code fragment to retrieve it

40% OBJECT-INTERACTION IDIOMS

21% VALID REFERENCES DETERMINING DEFINED STANDARD, OR FRAMEWORK IDENTIFIERS AT COMPILE TIME OR RUNTIME

OB1 *Inactionable Reference Error:*
statement generating error & error message → explanation of error message

OB2 *Silent Invalid Reference:*
invalid statement → warning message & statement fixing warning

16% BACK-END REQUESTS SENDING STRUCTURED DATA TO A SERVER, OR HANDLING SERVER RESPONSES

OB6 *Misconfigured Request:*
back-end request & desired behavior → modified request matching behavior

OB7 *Unclear Transmission:*
back-end request as sent → back-end request as received

OB8 *Mishandled Response:*
back-end request → code fragment for response(s) listening and parsing

8% SCOPE CONTEXTS IDENTIFYING THE CONTEXT GIVEN TO THE KEYWORD `this` WITHIN A CODE BLOCK, OR A VARIABLE'S VISIBILITY

OB12 *Unclear Scope:* `this` statement → owner scope of `this`

20% COLLECTIONS AND FORMATS CREATING OR MANIPULATING A COLLECTION, OR FORMATTING DATA FOR USE IN A FRAMEWORK OR LIBRARY

OB3 *Unidentified Iteration Construct:*
collection object → code fragment with corresponding iteration construct

OB4 *Occluded Modification:*
collection object & loop fragment → modifications of collection per iteration

OB5 *Confused Formatting:*
object in format A → code fragment converting object to format B

8% METHOD CHAINS DETERMINING THE EFFECTS OF A METHOD INVOCATION WITHIN A SEQUENCE OF CONSECUTIVE CALLS

OB9 *Incomplete Sequence:*
o.m1(…).m2(…)….mn(….) → o.m1(…).m2(…)….mk(…)….mn(…)

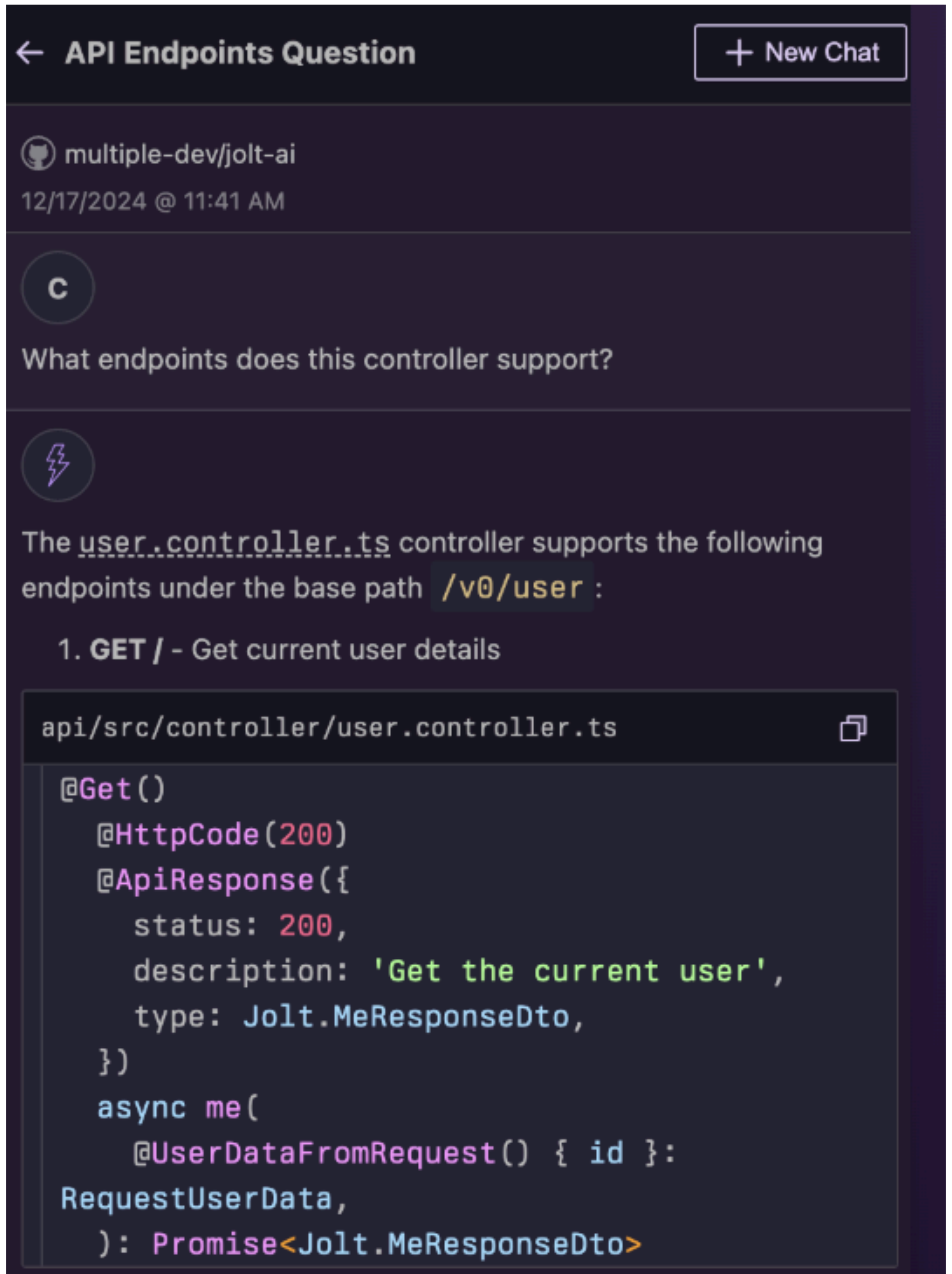
OB10 *Incorrect Sequence:*
o.m1(…).m2(…)….mn(….) → o.mk(…)….m1(…).mn(…)

OB11 *Overwritten Effect:*
o.m1(…).m2(…)….mn(….) → methods mk and ml where both mutate object

D. I. Samudio and T. D. LaToza, "Barriers in Front-End Web Development," 2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2022, pp. 1-11, doi: 10.1109/VL/HCC53370.2022.9833127.

Developers are already using LLMs to answer questions

- Alignment between questions developers ask and the questions LLMs can help answer determines support and improvement of program comprehension.
  - Which questions is it helping answer well already?
  - Where is it still struggling or less effective? (these questions become more important since still hard)



multiple-dev/jolt-ai
12/17/2024 @ 11:41 AM

C

What endpoints does this controller support?

The user.controller.ts controller supports the following endpoints under the base path `/v0/user`:

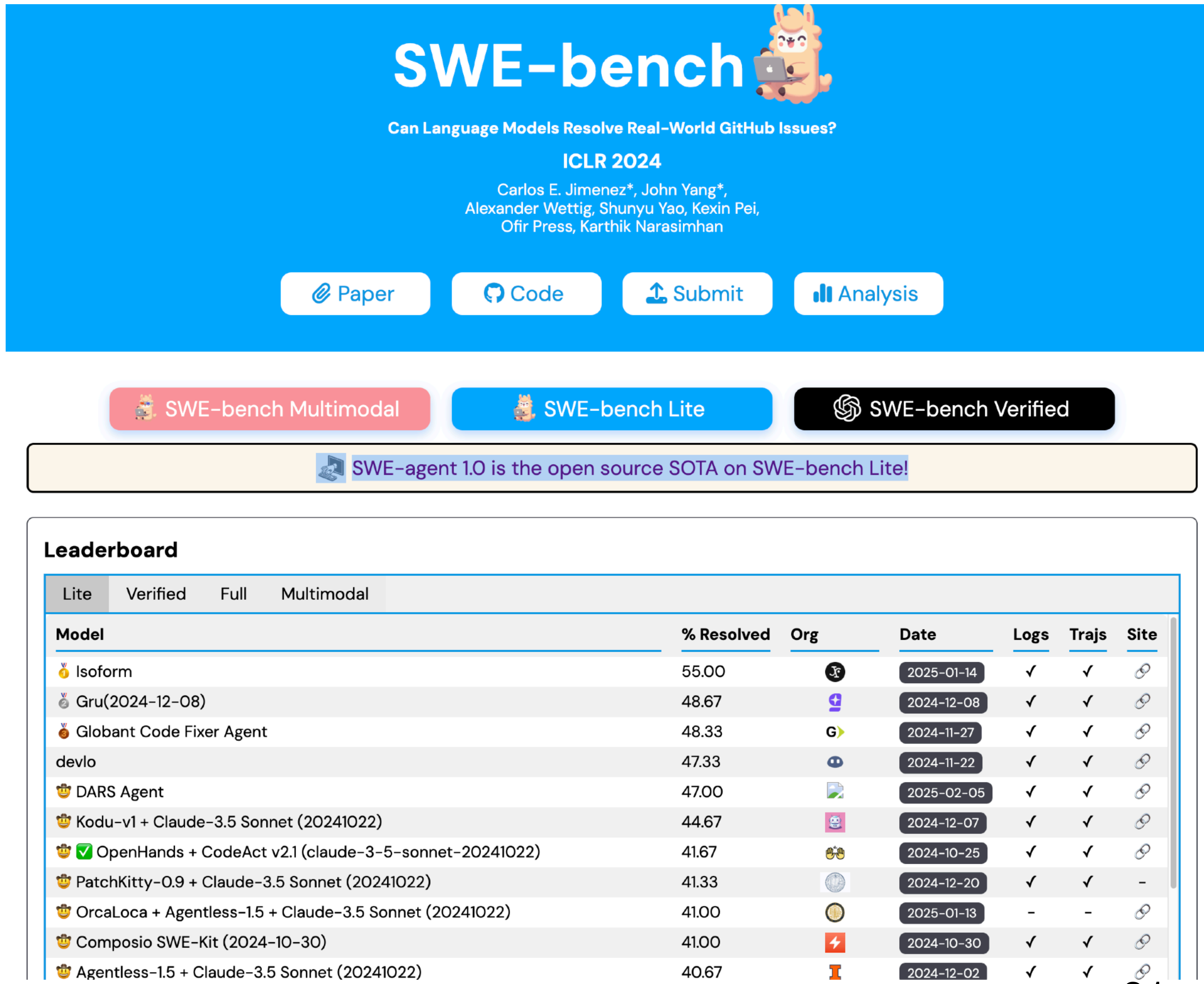
1. **GET /** - Get current user details

api/src/controller/user.controller.ts

```
@Get()
  @HttpCode(200)
  @ApiResponse({
    status: 200,
    description: 'Get the current user',
    type: Jolt.MeResponseDto,
  })
  async me(
    @UserDataFromRequest() { id }:
RequestUserData,
  ): Promise<Jolt.MeResponseDto>
```

Benchmarking Program Comprehension with LLMs

- Existing benchmarks examine full automation of SE (e.g., success in generating a patch)
  - Many tasks will still require developer involvement.
  - Want to have benchmarks that measure benefit to program comprehension of an LLM assistant
  - Can use hard to answer questions as benchmark



SWE-bench

Can Language Models Resolve Real-World GitHub Issues?

ICLR 2024

Carlos E. Jimenez*, John Yang*,
Alexander Wettig, Shunyu Yao, Kexin Pei,
Ofir Press, Karthik Narasimhan

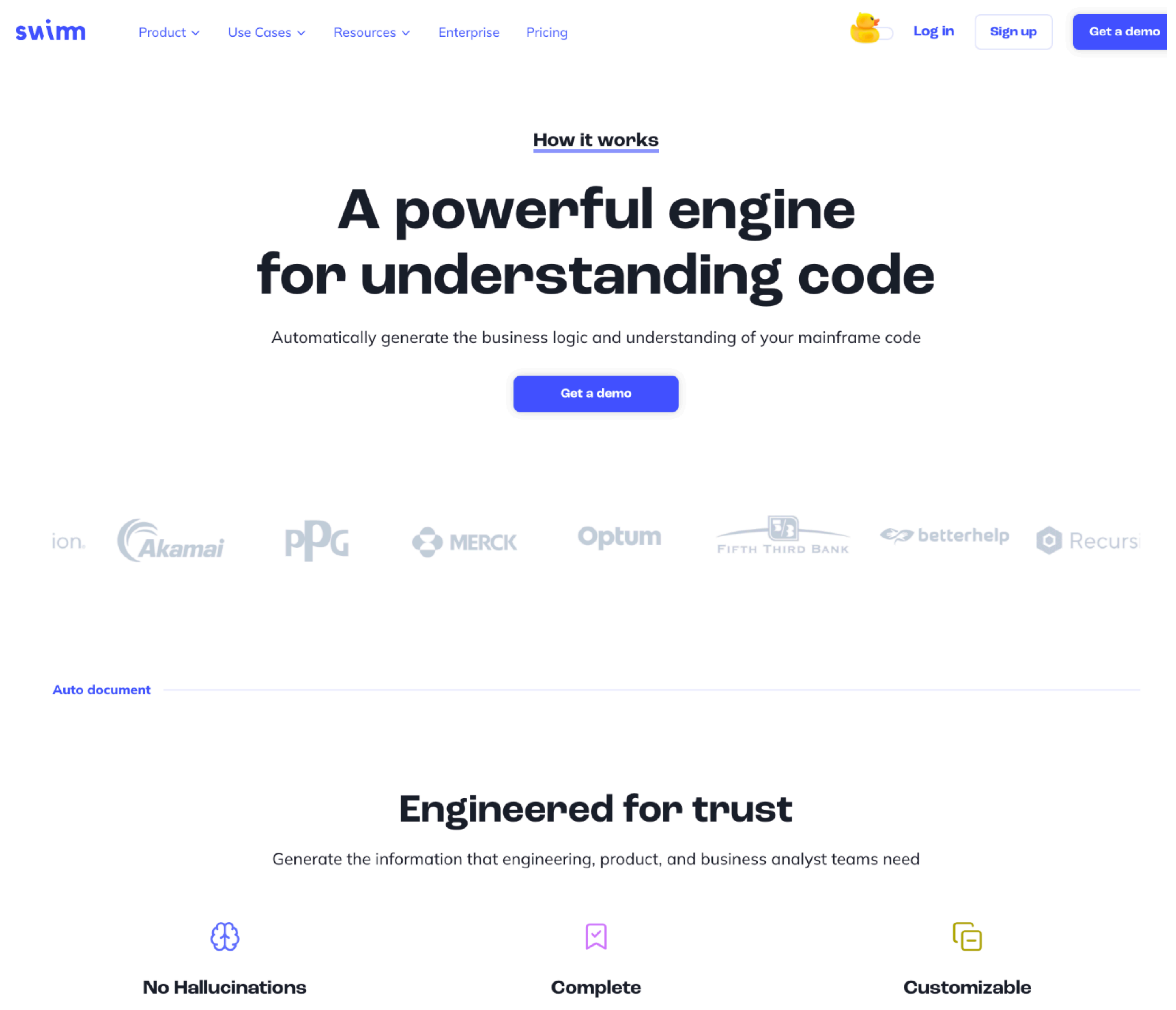
SWE-agent 1.0 is the open source SOTA on SWE-bench Lite!

Leaderboard

Model	% Resolved	Org	Date	Logs	Trajs	Site
🥇 Isoform	55.00		2025-01-14	✓	✓	🔗
🥈 Gru(2024-12-08)	48.67		2024-12-08	✓	✓	🔗
🥉 Globant Code Fixer Agent	48.33		2024-11-27	✓	✓	🔗
devlo	47.33		2024-11-22	✓	✓	🔗
🤠 DARS Agent	47.00		2025-02-05	✓	✓	🔗
🤠 Kodu-v1 + Claude-3.5 Sonnet (20241022)	44.67		2024-12-07	✓	✓	🔗
🤠 ✅ OpenHands + CodeAct v2.1 (claude-3-5-sonnet-20241022)	41.67		2024-10-25	✓	✓	🔗
🤠 PatchKitty-0.9 + Claude-3.5 Sonnet (20241022)	41.33		2024-12-20	✓	✓	-
🤠 OrcaLoca + Agentless-1.5 + Claude-3.5 Sonnet (20241022)	41.00		2025-01-13	-	-	🔗
🤠 Composio SWE-Kit (2024-10-30)	41.00		2024-10-30	✓	✓	🔗
🤠 Agentless-1.5 + Claude-3.5 Sonnet (20241022)	40.67		2024-12-02	✓	✓	🔗

LLMs and Organization Knowledge

- Big shift underway from using StackOverflow and other crowdsourced knowledge repos to using LLMs trained on these resources
- Internal dev tool orgs looking to promote knowledge sharing by connecting communication channels (issue trackers, Slack, design wikis, etc.) to LLMs
- If expertise all consumed through LLM, less motivation to document it explicitly?



A powerful engine for understanding code

Automatically generate the business logic and understanding of your mainframe code

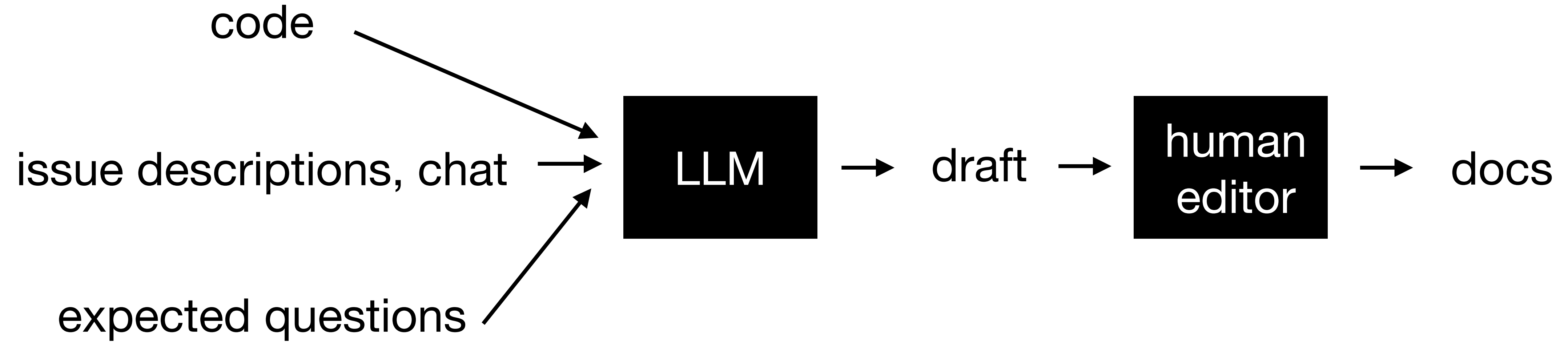
Engineered for trust

Generate the information that engineering, product, and business analyst teams need

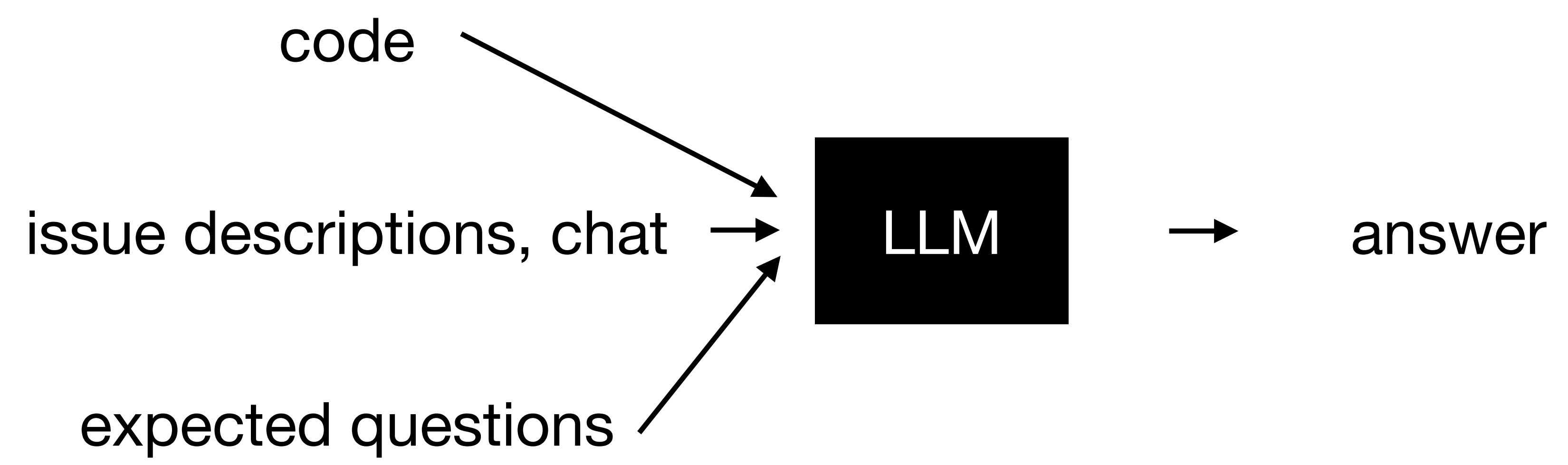
Documentation or Reverse Engineering?

- Should we build tools that create documentation?
  - Developer can see and approve documentation?
  - But what information is important enough to document?
  - And, given many questions are situational, how much can you really cover in the docs?
  - And can docs still go out of date?
- Or build tools that reverse engineer code to answer questions?
  - Can tackle any question
  - But how to ensure trust in the answer, if there is no developer signing off on them?

Documentation Generation



Reverse Engineering



Theories of Program Comprehension in the Age of LLMs

- How can developers still understand the code being generated?

  ⇒ Theories of Information Needs in Programming

- **To what extent do developers really have to understand the code being written?**

  ⇒ **Theories of Information Hiding**

- How do developers figure out what's wrong when it doesn't work?

  ⇒ Theories of Debugging

Theories of Information Hiding

- L**imit** information developers need to be aware of about code

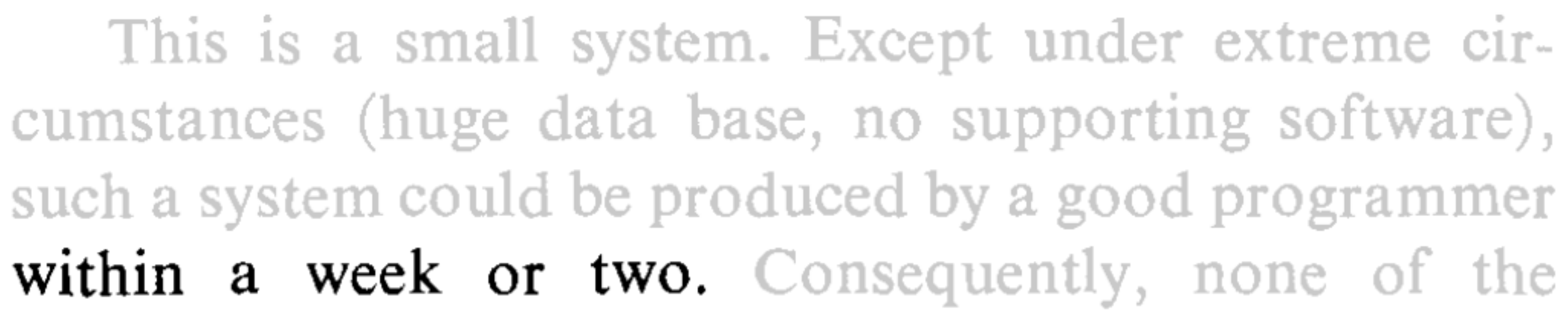
Theories of Information Hiding

- Abstraction - only think about the high-level operations of what some code does, not all the details
- Design by contract - don't need to understand the implementation, just the input/ouput behavior
- Information hiding - only the person writing the library / framework really needs to know all the details about how it works

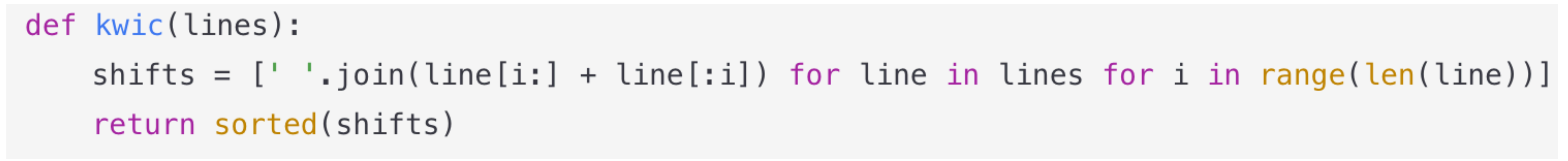
- Enable reuse - don't write the same old code again, just reuse a library or framework that does it

Powerful abstractions help build more quickly

- Parnas' **Key Words in Context** Problem, used to illustrate 1972 paper on information hiding
  - The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWXC index system outputs a listing of all circular shifts of all lines in alphabetical order.


This is a small system. Except under extreme circumstances (huge data base, no supporting software), such a system could be produced by a good programmer within a week or two. Consequently, none of the

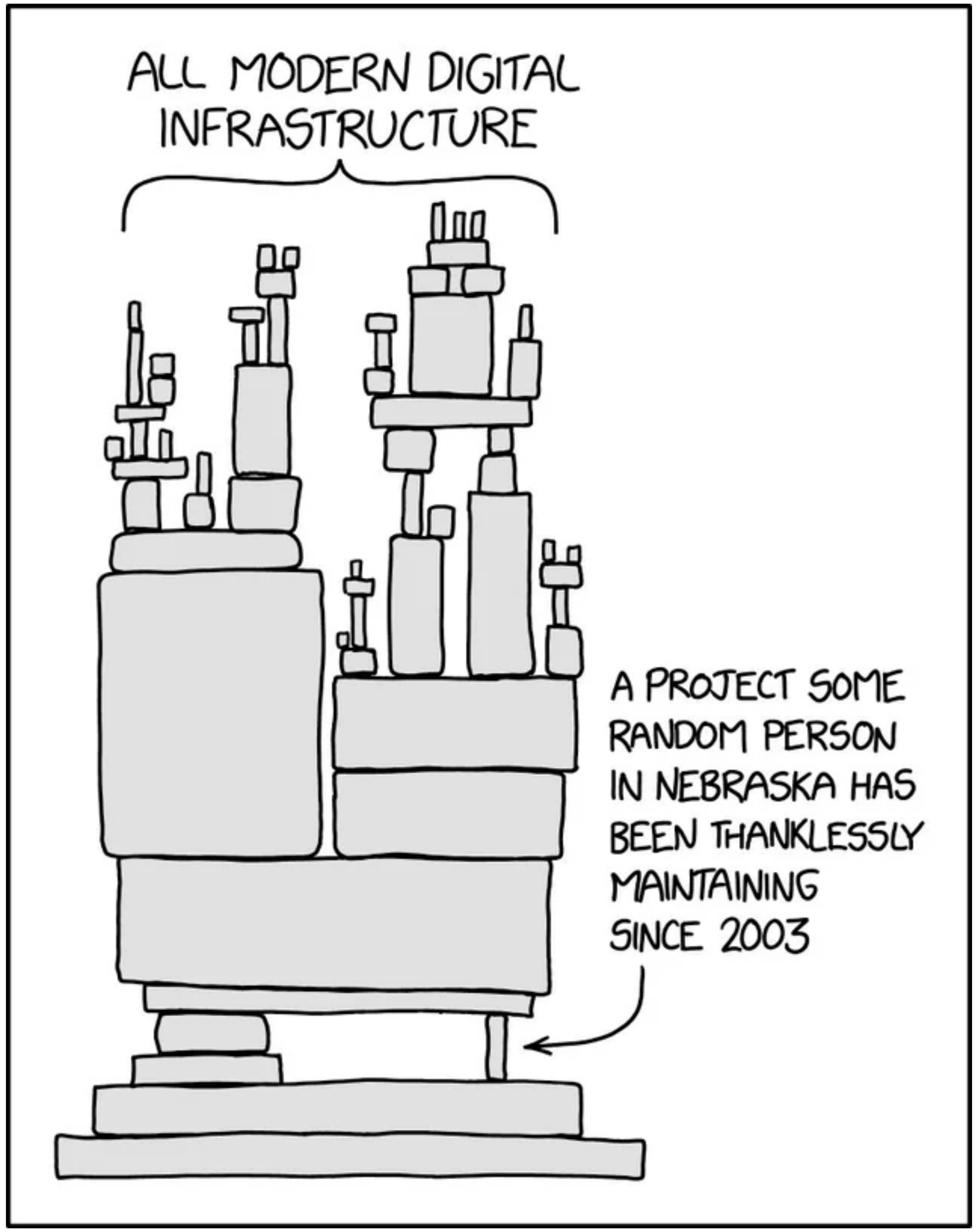
1972


```python
def kwic(lines):
    shifts = [' '.join(line[i:] + line[:i]) for line in lines for i in range(len(line))]
    return sorted(shifts)
```

2025

What happened???!?

- Modern collections libraries do almost all the work, with only a tiny bit of code written on top
- Much of the code that developers used to constantly rewrite, from scratch, is already written by someone else, stored in a library / framework somewhere


ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

xkcd 2347: Dependency

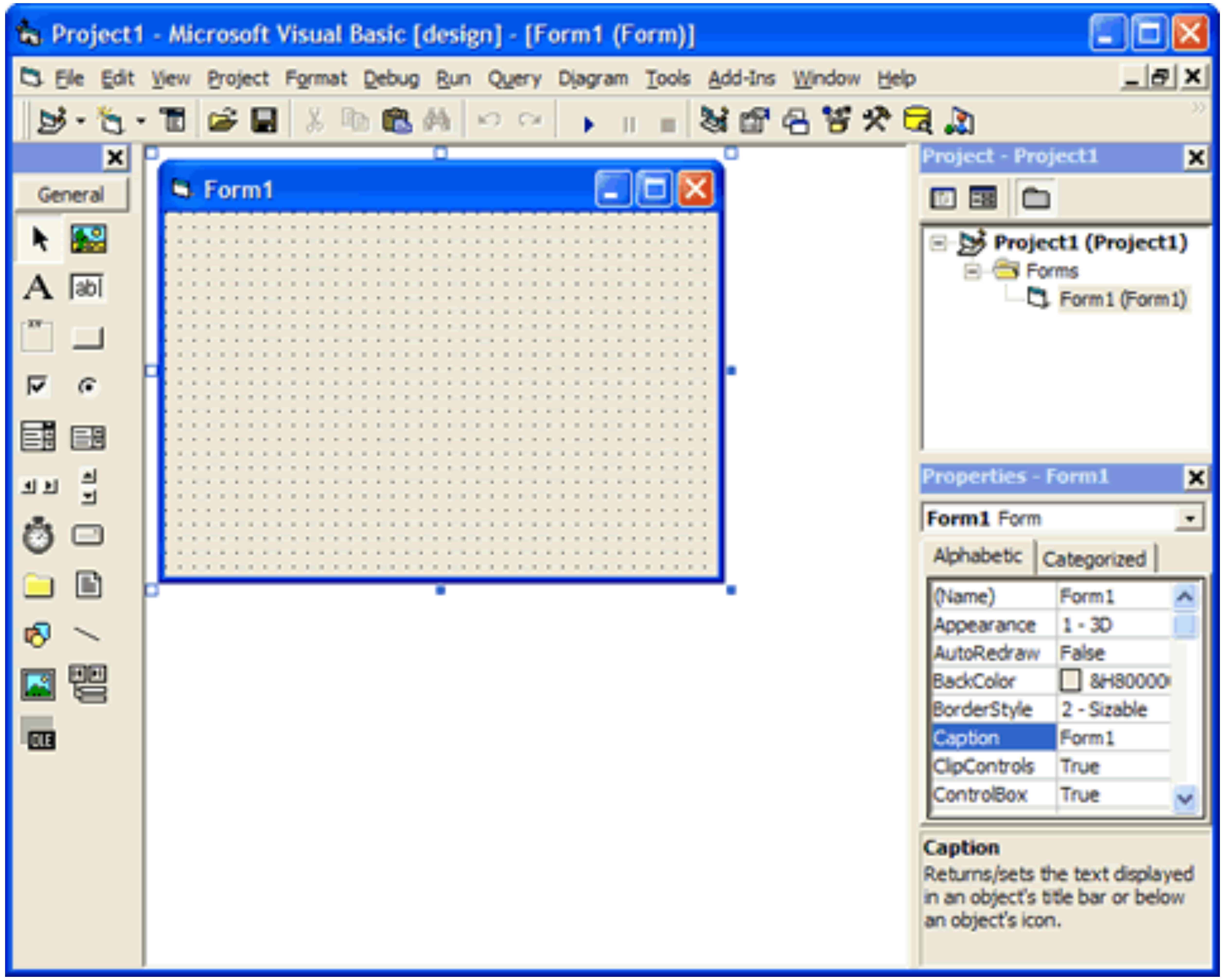
Limitations of abstraction

- Leaky abstractions: details that were supposed to be hidden still matter, particularly for qualities like performance
- Hidden dependencies: implementation may interact with other modules in unexpected ways, creating unexpected behavior, and making debugging difficult
- Hyrum's law
    - With a sufficient number of users of an API,
    - it does not matter what you promise in the contract:
    - all observable behaviors of your system
    - will be depended on by somebody.

Spolsky, Joel (2002). "The Law of Leaky Abstractions".

https://www.hyrumslaw.com/

Generating code with a tool



Project1 - Microsoft Visual Basic [design] - [Form1 (Form)]
File Edit View Project Format Debug Run Query Diagram Tools Add-Ins Window Help
General
Form1
Project - Project1
Project1 (Project1)
Forms
Form1 (Form1)
Properties - Form1
Form1 Form
Alphabetic
Categorized

(Name)	Form1
Appearance	1 - 3D
AutoRedraw	False
BackColor	&H80000
BorderStyle	2 - Sizable
Caption	Form1
ClipControls	True
ControlBox	True

Caption
Returns/sets the text displayed in an object's title bar or below an object's icon.

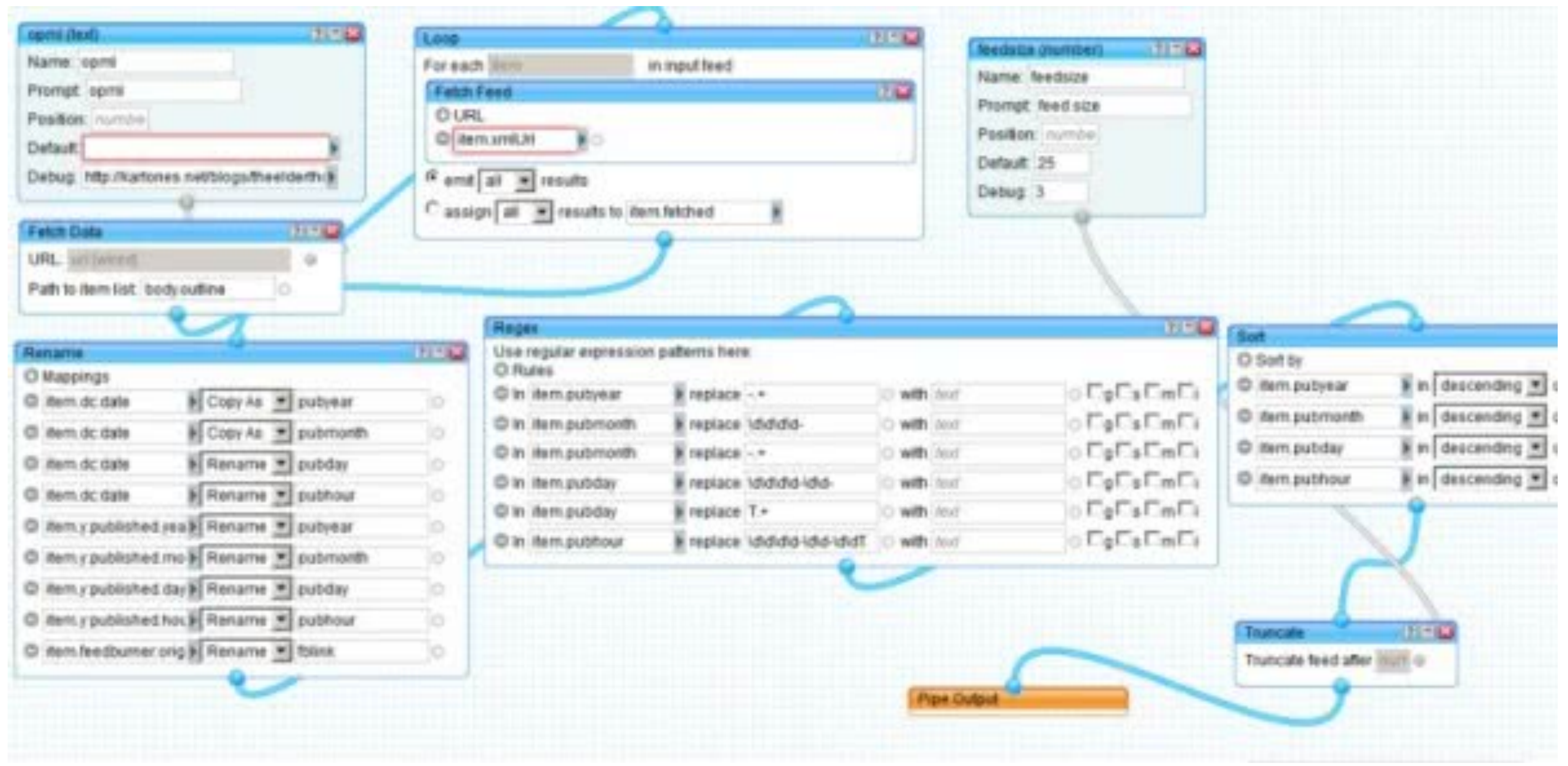
Theories of Abstraction in the age of LLMs

- Developers long told not to think too much about all the code they reuse
  - Trust the framework / library developers who wrote it that it works
- Developers vibe coding with an LLM generate lots of code that they also don't think too much about
  - But bad to trust the code too much, as the LLM makes more mistakes
- But....
  - What if most of the code IS framework code, already written before?

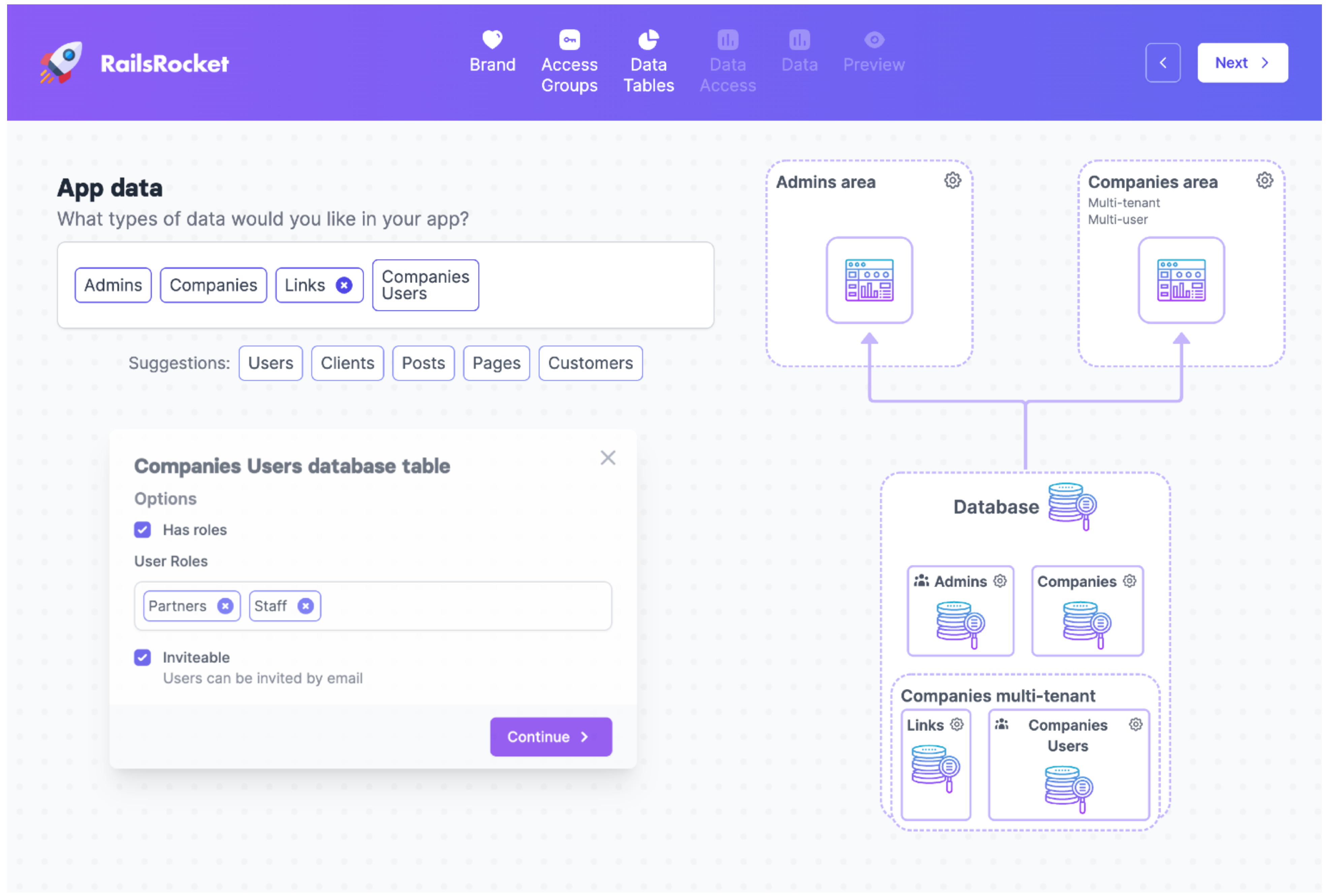
Theories of Abstraction in the Age of LLMs

- Small teams of expert software engineers build the hard building blocks that require deep system expertise, exposed as framework and libraries
- Everyone else vibe codes with LLMs to build small ephemeral apps on top of the underlying capabilities of LLMs

Continues trend of empowerment of EUP



opml (text)
Name: opml
Prompt: opml
Position: number
Default:
Debug: http://kartones.net/blogs/theelderlin
Fetch Data
URL:
Path to item list: body.outline
Loop
For each item in input feed
Fetch Feed
URL
item.xmlUrl
emit all results
assign all results to item.fetched
feedsize (number)
Name: feedsize
Prompt: feed size
Position: number
Default: 25
Debug: 3
Rename
Mappings
item.dc:date Copy As pubyear
item.dc:date Copy As pubmonth
item.dc:date Rename pubday
item.dc:date Rename pubhour
item.y:published.year Rename pubyear
item.y:published.month Rename pubmonth
item.y:published.day Rename pubday
item.y:published.hour Rename pubhour
item.feedburner:origLink Rename flink
Regex
Use regular expression patterns here
Rules
In item.pubyear replace -.* with
In item.pubmonth replace \d\d\d\d- with
In item.pubmonth replace -.* with
In item.pubday replace \d\d\d\d-\d\d- with
In item.pubday replace T.* with
In item.pubhour replace \d\d\d\d-\d\d-\d\dT with
Sort
Sort by
item.pubyear in descending
item.pubmonth in descending
item.pubday in descending
item.pubhour in descending
Truncate
Truncate feed after
Pipe Output



App data

What types of data would you like in your app?

Admins | Companies | Links | Companies Users

Suggestions: Users | Clients | Posts | Pages | Customers

Companies Users database table

Options

☑ Has roles

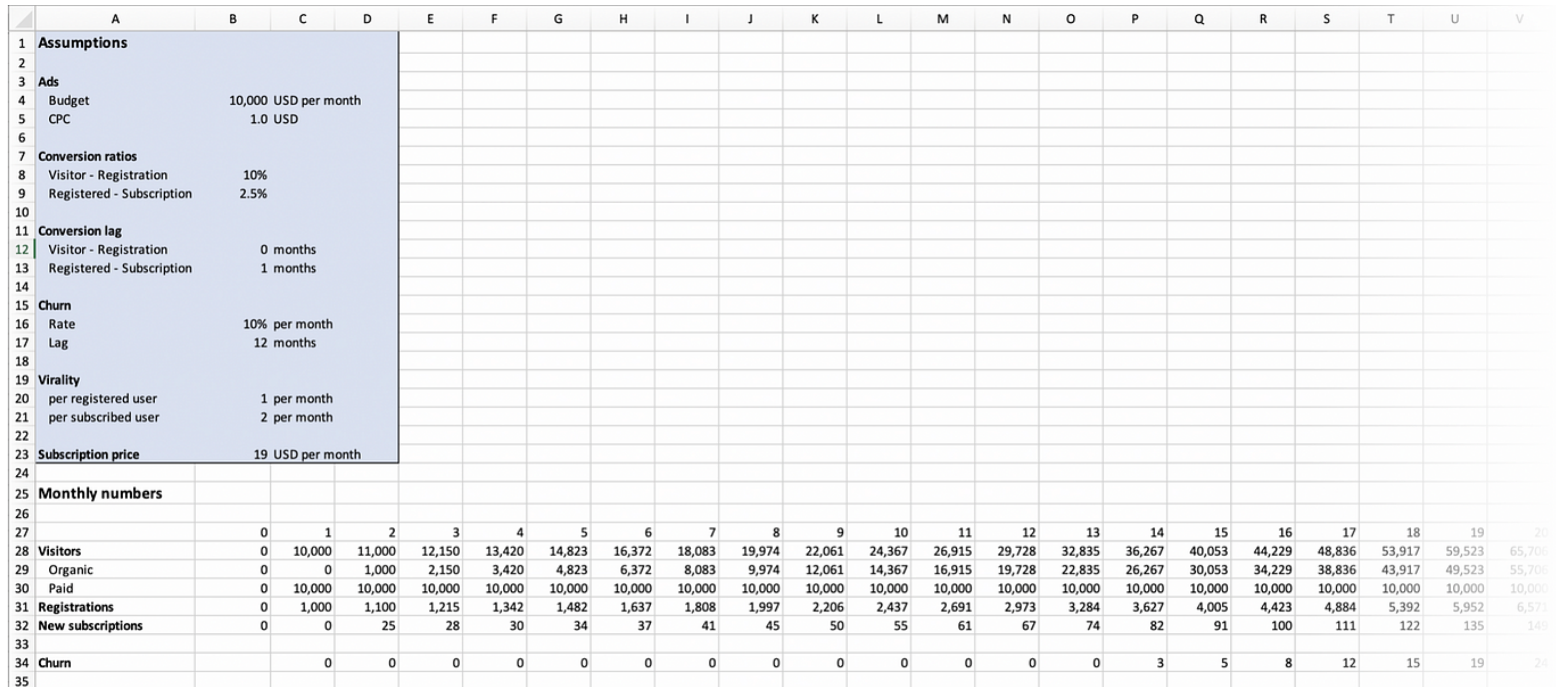
User Roles

Partners | Staff

☑ Inviteable

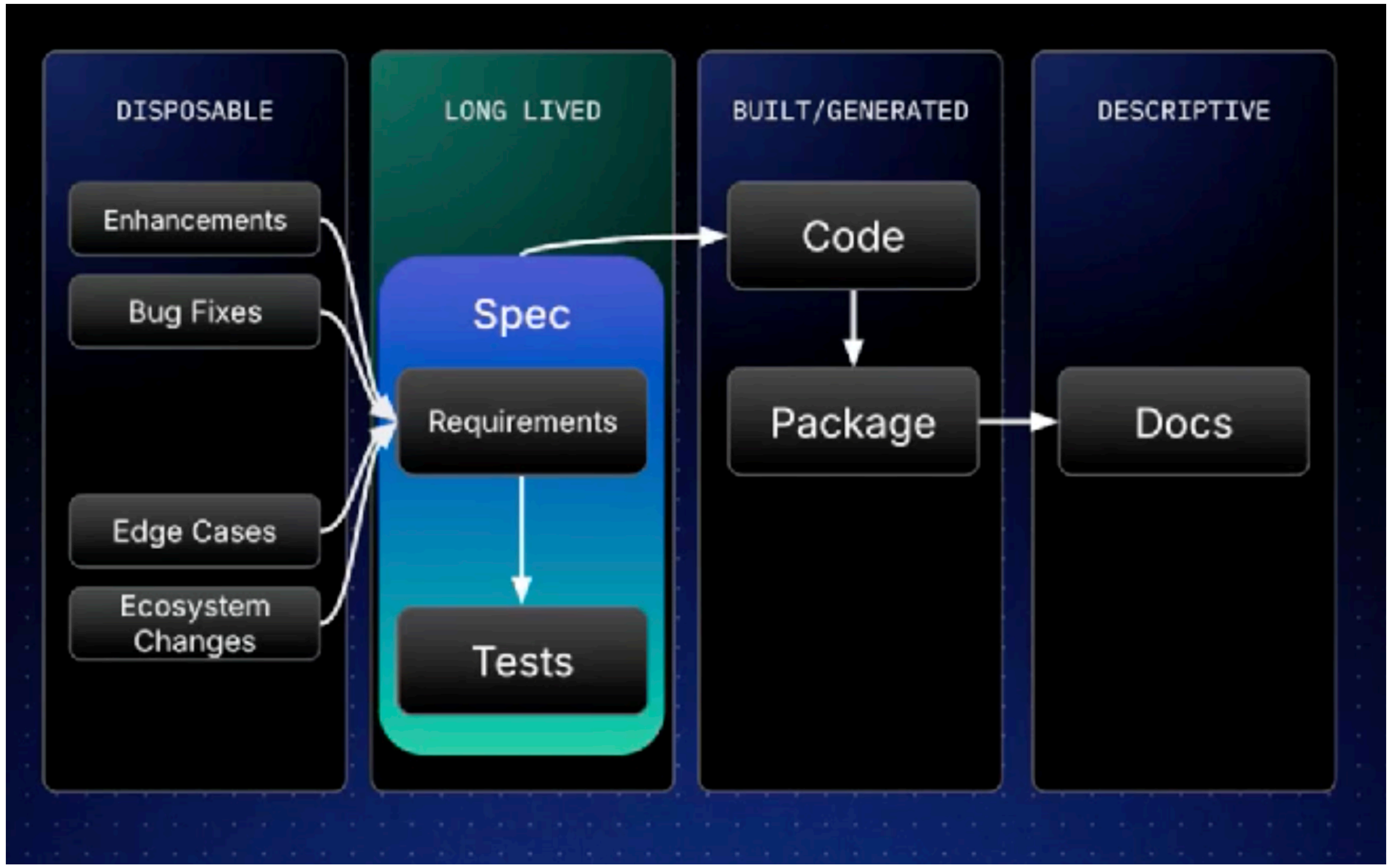
Users can be invited by email

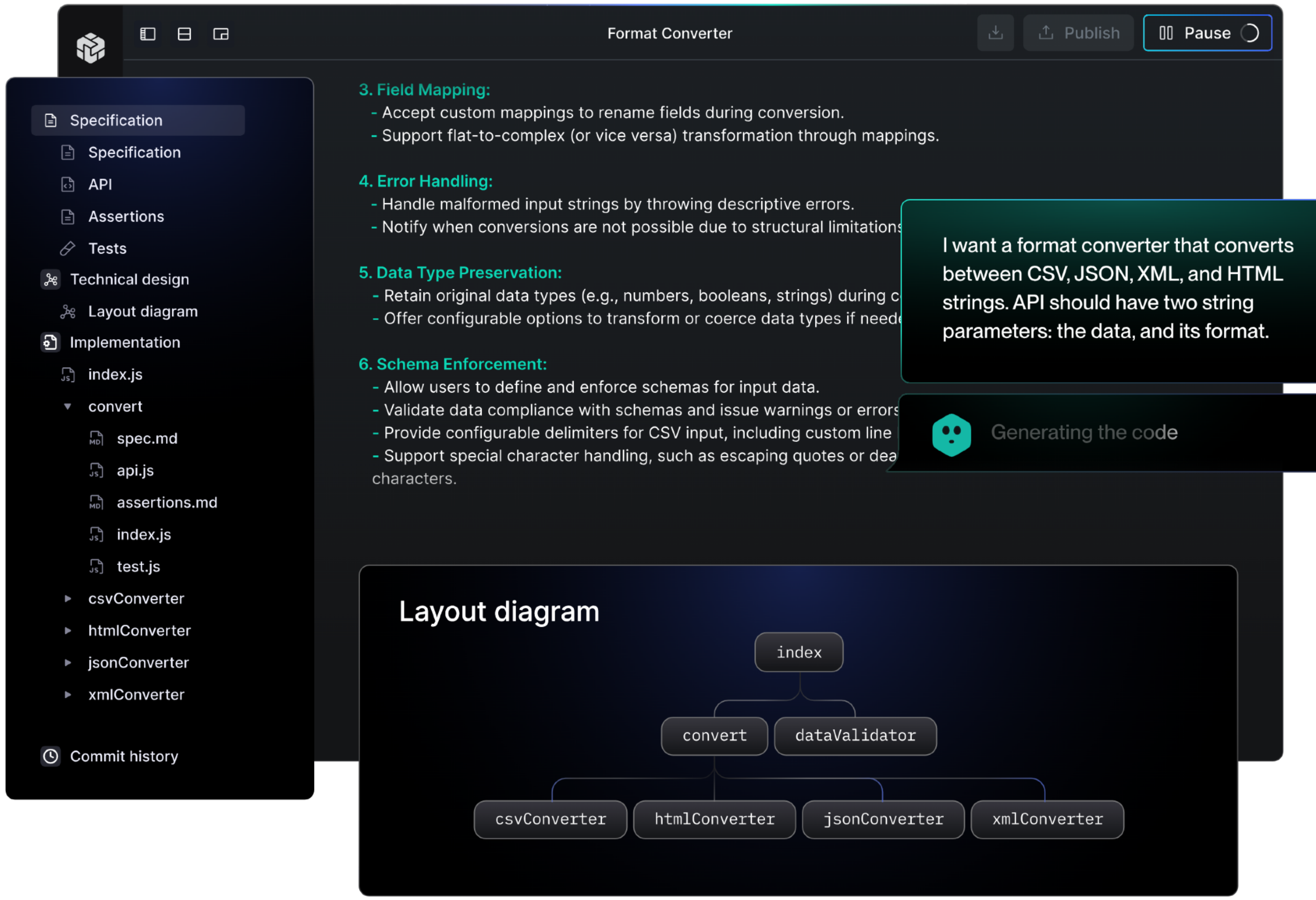
Continue



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	**Assumptions**																					
2																						
3	**Ads**																					
4	Budget	10,000	USD per month																			
5	CPC	1.0	USD																			
6																						
7	**Conversion ratios**																					
8	Visitor - Registration	10%																				
9	Registered - Subscription	2.5%																				
10																						
11	**Conversion lag**																					
12	Visitor - Registration	0	months																			
13	Registered - Subscription	1	months																			
14																						
15	**Churn**																					
16	Rate	10%	per month																			
17	Lag	12	months																			
18																						
19	**Virality**																					
20	per registered user	1	per month																			
21	per subscribed user	2	per month																			
22																						
23	**Subscription price**	19	USD per month																			
24																						
25	**Monthly numbers**																					
26																						
27		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
28	**Visitors**	0	10,000	11,000	12,150	13,420	14,823	16,372	18,083	19,974	22,061	24,367	26,915	29,728	32,835	36,267	40,053	44,229	48,836	53,917	59,523	65,706
29	Organic	0	0	1,000	2,150	3,420	4,823	6,372	8,083	9,974	12,061	14,367	16,915	19,728	22,835	26,267	30,053	34,229	38,836	43,917	49,523	55,706
30	Paid	0	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000	10,000
31	**Registrations**	0	1,000	1,100	1,215	1,342	1,482	1,637	1,808	1,997	2,206	2,437	2,691	2,973	3,284	3,627	4,005	4,423	4,884	5,392	5,952	6,571
32	**New subscriptions**	0	0	25	28	30	34	37	41	45	50	55	61	67	74	82	91	100	111	122	135	149
33																						
34	**Churn**		0	0	0	0	0	0	0	0	0	0	0	0	0	3	5	8	12	15	19	24
35																						

AI Native Spec-Driven Development





3. Field Mapping:
- Accept custom mappings to rename fields during conversion.
- Support flat-to-complex (or vice versa) transformation through mappings.

4. Error Handling:
- Handle malformed input strings by throwing descriptive errors.
- Notify when conversions are not possible due to structural limitations

5. Data Type Preservation:
- Retain original data types (e.g., numbers, booleans, strings) during c
- Offer configurable options to transform or coerce data types if need

6. Schema Enforcement:
- Allow users to define and enforce schemas for input data.
- Validate data compliance with schemas and issue warnings or errors
- Provide configurable delimiters for CSV input, including custom line
- Support special character handling, such as escaping quotes or dea
characters.

I want a format converter that converts between CSV, JSON, XML, and HTML strings. API should have two string parameters: the data, and its format.

https://www.tessl.io/

Assuring LLM interactions

- LLMs hallucinate and sometimes create bad solutions
  - What techniques do developers use to build trust in code?
    - Functional correctness: unit tests
    - Quality attributes (performance, extensibility, maintainability, scalability, ...): ???
  - How can developers have more visibility, control, and traceability across what LLMs are doing?

Theories of Program Comprehension in the Age of LLMs

- How can developers still understand the code being generated?

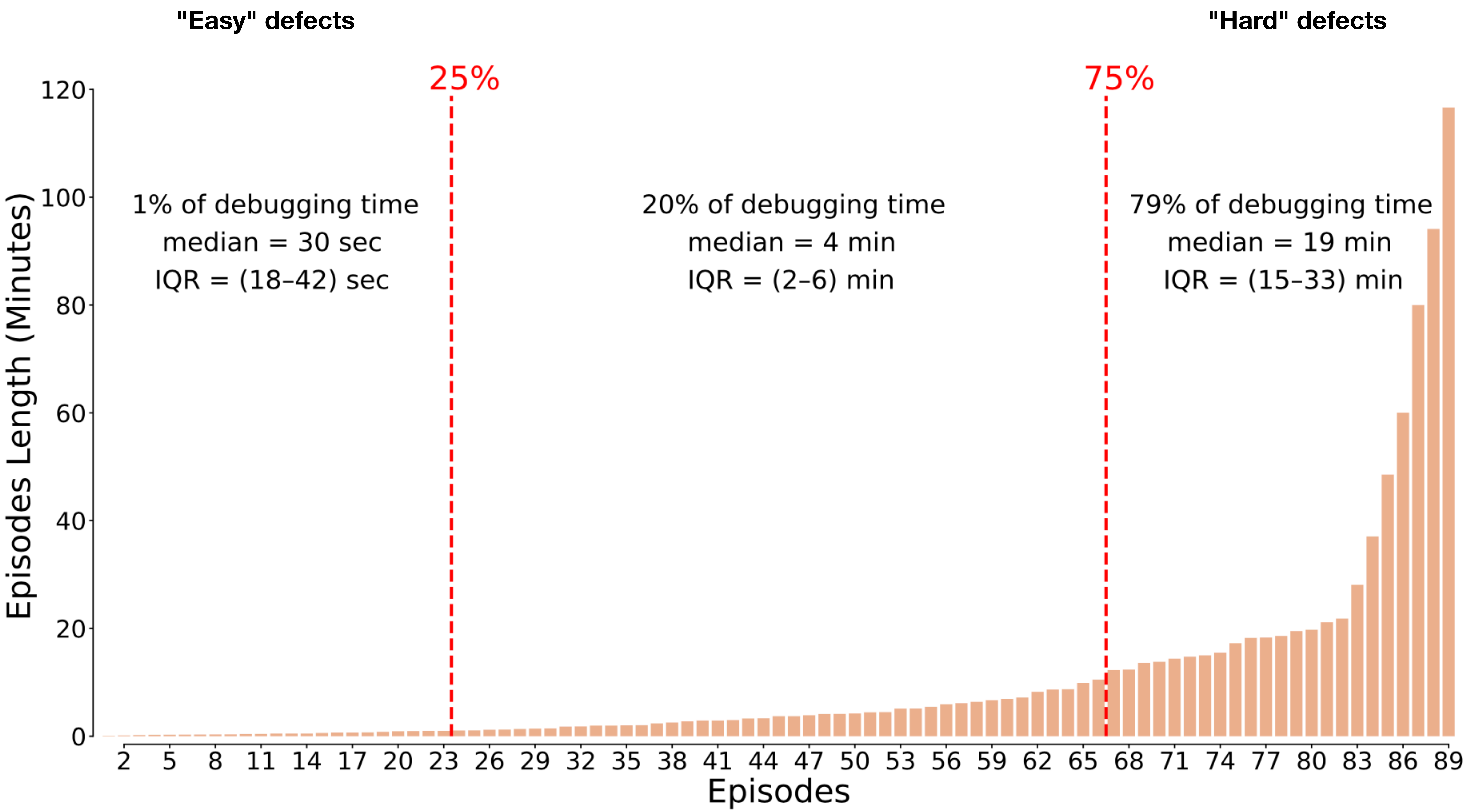
  ⇒ Theories of Information Needs in Programming

- To what extent do developers really have to understand the code being written?

  ⇒ Theories of Information Hiding

- **How do developers figure out what's wrong when it doesn't work?**

  ⇒ Theories of Debugging

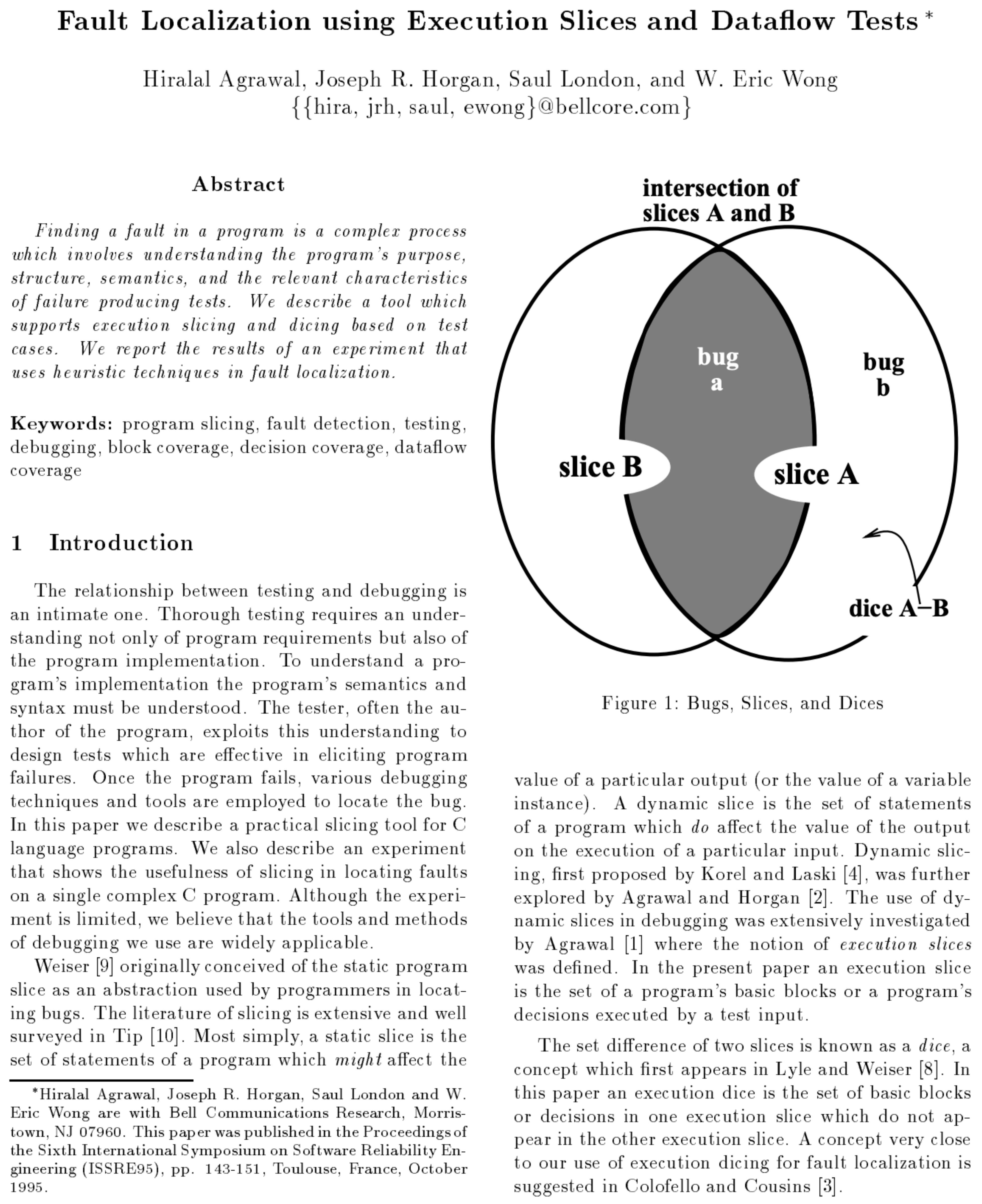

"Easy" defects
"Hard" defects
25%
75%
1% of debugging time
median = 30 sec
IQR = (18–42) sec
20% of debugging time
median = 4 min
IQR = (2–6) min
79% of debugging time
median = 19 min
IQR = (15–33) min
Episodes Length (Minutes)
0
20
40
60
80
100
120
2
5
8
11
14
17
20
23
26
29
32
35
38
41
44
47
50
53
56
59
62
65
68
71
74
77
80
83
86
89
Episodes

Theories of Debugging

- Fault localization: debugging is the process of finding the line with the defect
- Slicing: navigate forwards or backwards across control & data dependencies to locate the defective line
- Strategies: follow a strategy to investigate the code in a systematic way
- Hypothesis testing: use intuition to form hypothesis and sytematically gather evidence to accept or reject

Fault Localization

- Developers debugging first locate the statement with the fault.
- Automatic fault localization supports debugging by showing developers this line.



Fault Localization using Execution Slices and Dataflow Tests [*]

Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong
{{hira, jrh, saul, ewong}@bellcore.com}

Abstract

Finding a fault in a program is a complex process which involves understanding the program's purpose, structure, semantics, and the relevant characteristics of failure producing tests. We describe a tool which supports execution slicing and dicing based on test cases. We report the results of an experiment that uses heuristic techniques in fault localization.

Keywords: program slicing, fault detection, testing, debugging, block coverage, decision coverage, dataflow coverage

1 Introduction

The relationship between testing and debugging is an intimate one. Thorough testing requires an understanding not only of program requirements but also of the program implementation. To understand a program's implementation the program's semantics and syntax must be understood. The tester, often the author of the program, exploits this understanding to design tests which are effective in eliciting program failures. Once the program fails, various debugging techniques and tools are employed to locate the bug. In this paper we describe a practical slicing tool for C language programs. We also describe an experiment that shows the usefulness of slicing in locating faults on a single complex C program. Although the experiment is limited, we believe that the tools and methods of debugging we use are widely applicable.

Weiser [9] originally conceived of the static program slice as an abstraction used by programmers in locating bugs. The literature of slicing is extensive and well surveyed in Tip [10]. Most simply, a static slice is the set of statements of a program which *might* affect the value of a particular output (or the value of a variable instance). A dynamic slice is the set of statements of a program which *do* affect the value of the output on the execution of a particular input. Dynamic slicing, first proposed by Korel and Laski [4], was further explored by Agrawal and Horgan [2]. The use of dynamic slices in debugging was extensively investigated by Agrawal [1] where the notion of *execution slices* was defined. In the present paper an execution slice is the set of a program's basic blocks or a program's decisions executed by a test input.

The set difference of two slices is known as a *dice*, a concept which first appears in Lyle and Weiser [8]. In this paper an execution dice is the set of basic blocks or decisions in one execution slice which do not appear in the other execution slice. A concept very close to our use of execution dicing for fault localization is suggested in Colofello and Cousins [3].

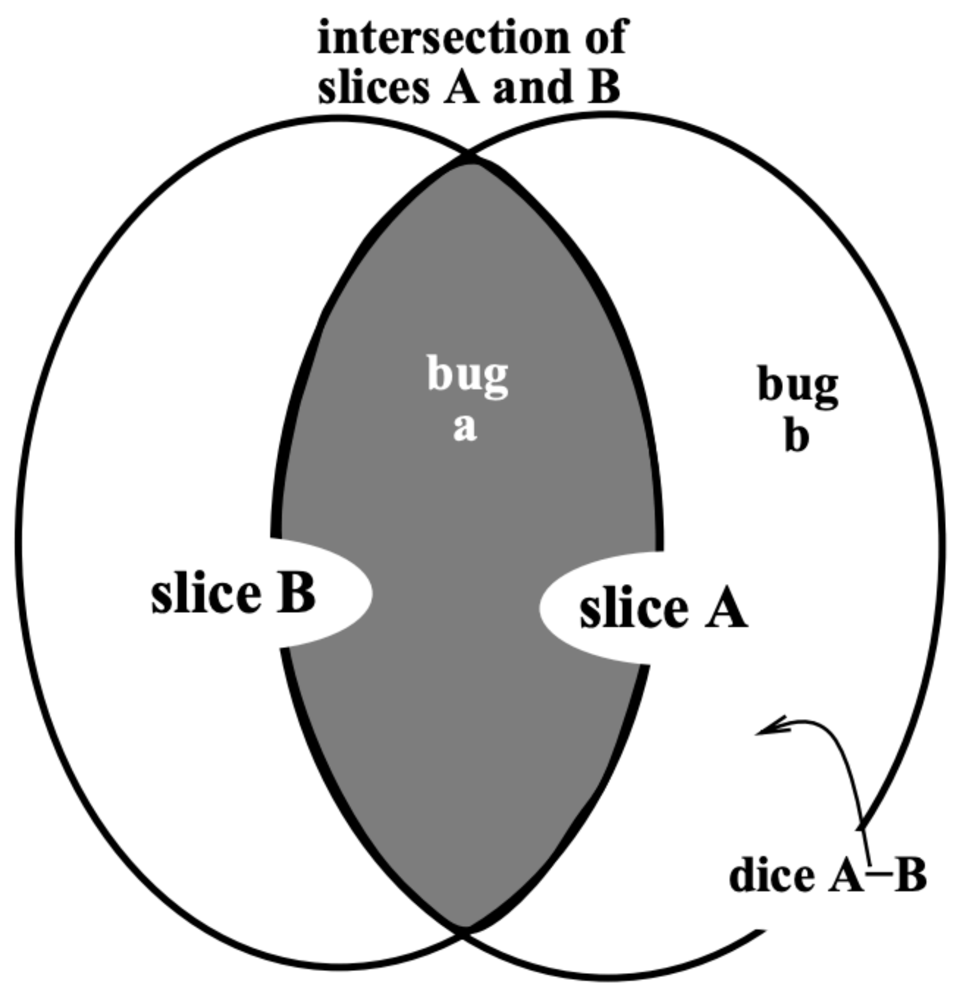
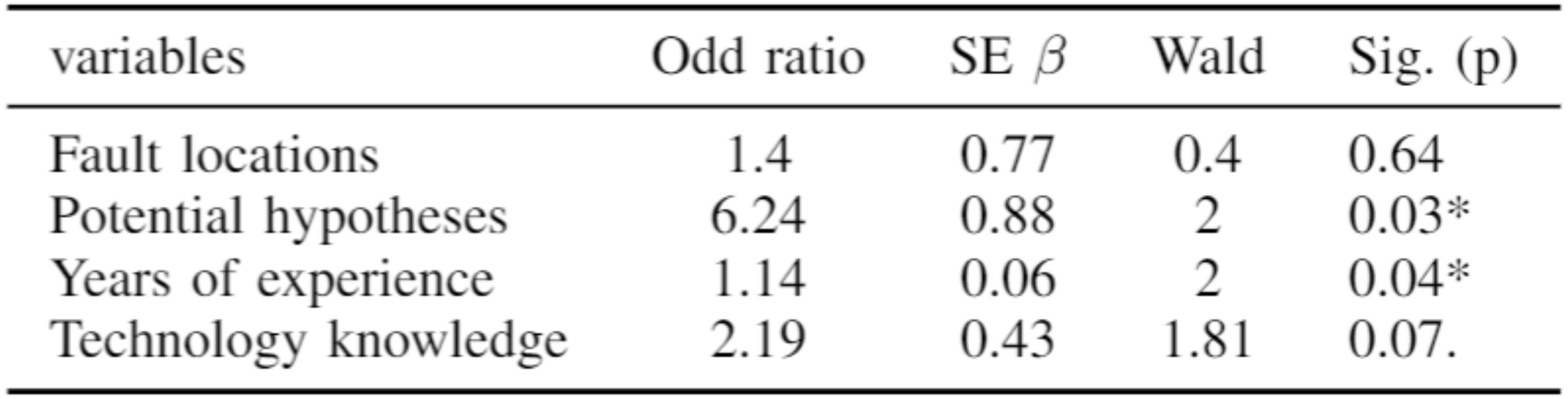


Figure 1: Bugs, Slices, and Dices

[*]Hiralal Agrawal, Joseph R. Horgan, Saul London and W. Eric Wong are with Bell Communications Research, Morristown, NJ 07960. This paper was published in the Proceedings of the Sixth International Symposium on Software Reliability Engineering (ISSRE95), pp. 143-151, Toulouse, France, October 1995.

Limitations of Fault Localization

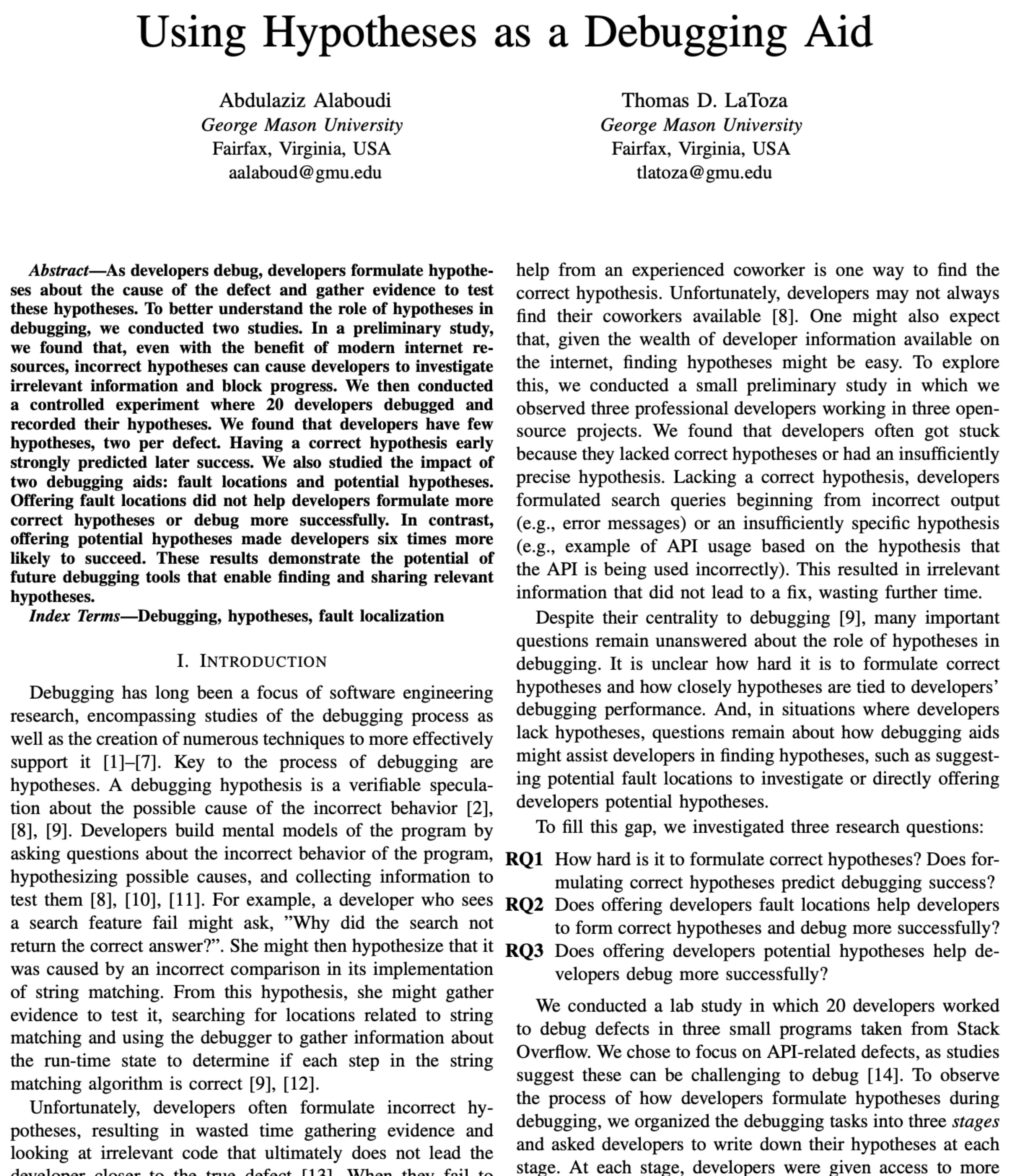
- Studies find that showing developers the line with the defect may not help



variables	Odd ratio	SE β	Wald	Sig. (p)
Fault locations	1.4	0.77	0.4	0.64
Potential hypotheses	6.24	0.88	2	0.03*
Years of experience	1.14	0.06	2	0.04*
Technology knowledge	2.19	0.43	1.81	0.07.

- Often need an **explanation** about **cause** of defect to explain why statement is incorrect

- Can only sometimes find the line w/ defect



Using Hypotheses as a Debugging Aid

Abdulaziz Alaboudi
George Mason University
Fairfax, Virginia, USA
aalaboud@gmu.edu

Thomas D. LaToza
George Mason University
Fairfax, Virginia, USA
tlatoza@gmu.edu

***Abstract*—As developers debug, developers formulate hypotheses about the cause of the defect and gather evidence to test these hypotheses. To better understand the role of hypotheses in debugging, we conducted two studies. In a preliminary study, we found that, even with the benefit of modern internet resources, incorrect hypotheses can cause developers to investigate irrelevant information and block progress. We then conducted a controlled experiment where 20 developers debugged and recorded their hypotheses. We found that developers have few hypotheses, two per defect. Having a correct hypothesis early strongly predicted later success. We also studied the impact of two debugging aids: fault locations and potential hypotheses. Offering fault locations did not help developers formulate more correct hypotheses or debug more successfully. In contrast, offering potential hypotheses made developers six times more likely to succeed. These results demonstrate the potential of future debugging tools that enable finding and sharing relevant hypotheses.**

***Index Terms*—Debugging, hypotheses, fault localization**

I. Introduction

Debugging has long been a focus of software engineering research, encompassing studies of the debugging process as well as the creation of numerous techniques to more effectively support it [1]–[7]. Key to the process of debugging are hypotheses. A debugging hypothesis is a verifiable speculation about the possible cause of the incorrect behavior [2], [8], [9]. Developers build mental models of the program by asking questions about the incorrect behavior of the program, hypothesizing possible causes, and collecting information to test them [8], [10], [11]. For example, a developer who sees a search feature fail might ask, "Why did the search not return the correct answer?". She might then hypothesize that it was caused by an incorrect comparison in its implementation of string matching. From this hypothesis, she might gather evidence to test it, searching for locations related to string matching and using the debugger to gather information about the run-time state to determine if each step in the string matching algorithm is correct [9], [12].

Unfortunately, developers often formulate incorrect hypotheses, resulting in wasted time gathering evidence and looking at irrelevant code that ultimately does not lead the developer closer to the true defect [13]. When they fail to

help from an experienced coworker is one way to find the correct hypothesis. Unfortunately, developers may not always find their coworkers available [8]. One might also expect that, given the wealth of developer information available on the internet, finding hypotheses might be easy. To explore this, we conducted a small preliminary study in which we observed three professional developers working in three open-source projects. We found that developers often got stuck because they lacked correct hypotheses or had an insufficiently precise hypothesis. Lacking a correct hypothesis, developers formulated search queries beginning from incorrect output (e.g., error messages) or an insufficiently specific hypothesis (e.g., example of API usage based on the hypothesis that the API is being used incorrectly). This resulted in irrelevant information that did not lead to a fix, wasting further time.

Despite their centrality to debugging [9], many important questions remain unanswered about the role of hypotheses in debugging. It is unclear how hard it is to formulate correct hypotheses and how closely hypotheses are tied to developers' debugging performance. And, in situations where developers lack hypotheses, questions remain about how debugging aids might assist developers in finding hypotheses, such as suggesting potential fault locations to investigate or directly offering developers potential hypotheses.

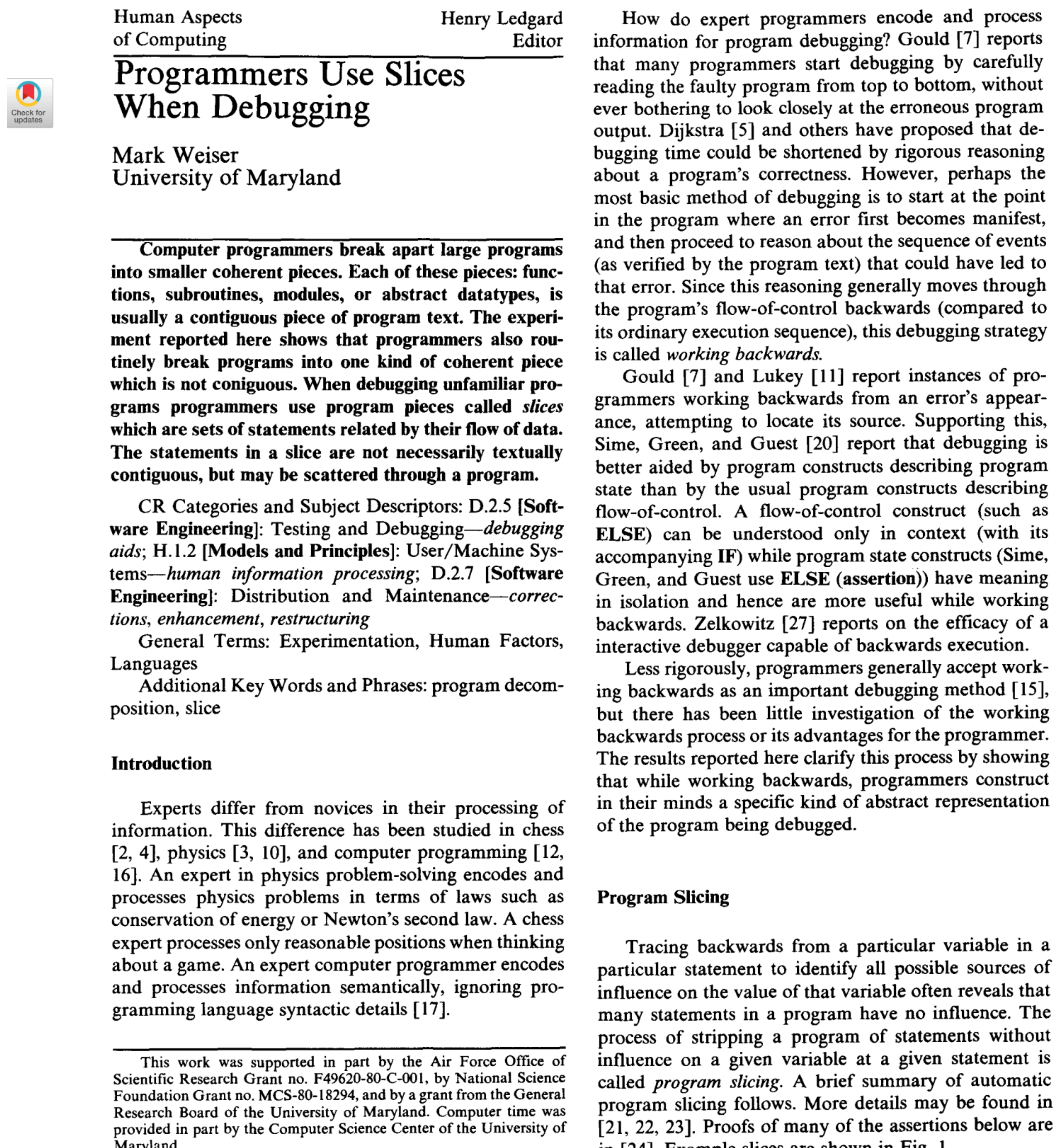
To fill this gap, we investigated three research questions:

RQ1 How hard is it to formulate correct hypotheses? Does formulating correct hypotheses predict debugging success?
RQ2 Does offering developers fault locations help developers to form correct hypotheses and debug more successfully?
RQ3 Does offering developers potential hypotheses help developers debug more successfully?

We conducted a lab study in which 20 developers worked to debug defects in three small programs taken from Stack Overflow. We chose to focus on API-related defects, as studies suggest these can be challenging to debug [14]. To observe the process of how developers formulate hypotheses during debugging, we organized the debugging tasks into three *stages* and asked developers to write down their hypotheses at each stage. At each stage, developers were given access to more

Slicing

- Developers start at output statement that generates symptom.
- Developers navigate control & data flow backwards & navigate across control & data flow backwards to understand



Human Aspects of Computing — Henry Ledgard, Editor

Programmers Use Slices When Debugging

Mark Weiser
University of Maryland

Computer programmers break apart large programs into smaller coherent pieces. Each of these pieces: functions, subroutines, modules, or abstract datatypes, is usually a contiguous piece of program text. The experiment reported here shows that programmers also routinely break programs into one kind of coherent piece which is not coniguous. When debugging unfamiliar programs programmers use program pieces called *slices* which are sets of statements related by their flow of data. The statements in a slice are not necessarily textually contiguous, but may be scattered through a program.

CR Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; H.1.2 [**Models and Principles**]: User/Machine Systems—*human information processing*; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*corrections, enhancement, restructuring*

General Terms: Experimentation, Human Factors, Languages

Additional Key Words and Phrases: program decomposition, slice

Introduction

Experts differ from novices in their processing of information. This difference has been studied in chess [2, 4], physics [3, 10], and computer programming [12, 16]. An expert in physics problem-solving encodes and processes physics problems in terms of laws such as conservation of energy or Newton's second law. A chess expert processes only reasonable positions when thinking about a game. An expert computer programmer encodes and processes information semantically, ignoring programming language syntactic details [17].

How do expert programmers encode and process information for program debugging? Gould [7] reports that many programmers start debugging by carefully reading the faulty program from top to bottom, without ever bothering to look closely at the erroneous program output. Dijkstra [5] and others have proposed that debugging time could be shortened by rigorous reasoning about a program's correctness. However, perhaps the most basic method of debugging is to start at the point in the program where an error first becomes manifest, and then proceed to reason about the sequence of events (as verified by the program text) that could have led to that error. Since this reasoning generally moves through the program's flow-of-control backwards (compared to its ordinary execution sequence), this debugging strategy is called *working backwards.*

Gould [7] and Lukey [11] report instances of programmers working backwards from an error's appearance, attempting to locate its source. Supporting this, Sime, Green, and Guest [20] report that debugging is better aided by program constructs describing program state than by the usual program constructs describing flow-of-control. A flow-of-control construct (such as **ELSE**) can be understood only in context (with its accompanying **IF**) while program state constructs (Sime, Green, and Guest use **ELSE (assertion)**) have meaning in isolation and hence are more useful while working backwards. Zelkowitz [27] reports on the efficacy of a interactive debugger capable of backwards execution.

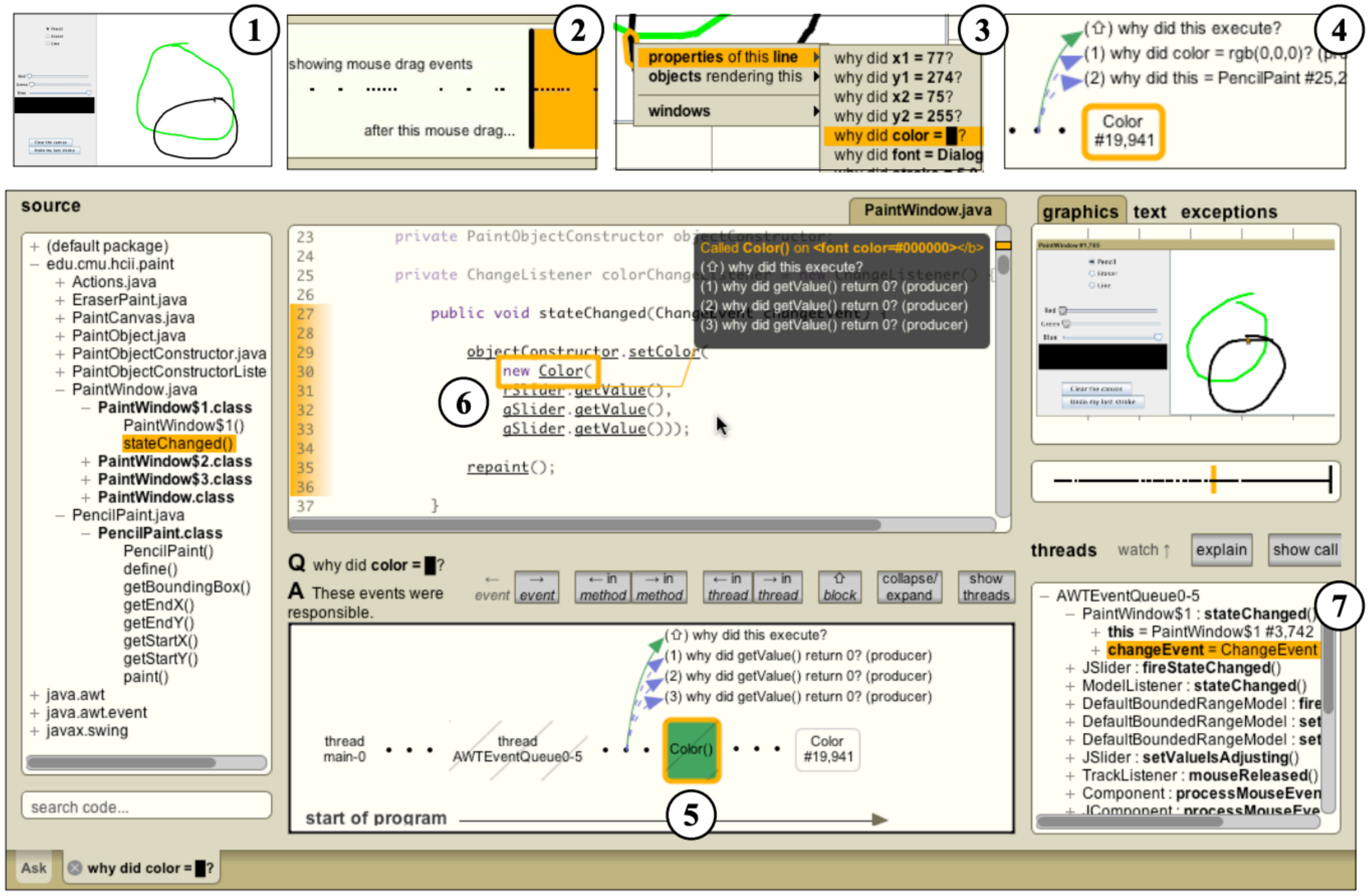
Less rigorously, programmers generally accept working backwards as an important debugging method [15], but there has been little investigation of the working backwards process or its advantages for the programmer. The results reported here clarify this process by showing that while working backwards, programmers construct in their minds a specific kind of abstract representation of the program being debugged.

Program Slicing

Tracing backwards from a particular variable in a particular statement to identify all possible sources of influence on the value of that variable often reveals that many statements in a program have no influence. The process of stripping a program of statements without influence on a given variable at a given statement is called *program slicing*. A brief summary of automatic program slicing follows. More details may be found in [21, 22, 23]. Proofs of many of the assertions below are in [24]. Example slices are shown in Fig. 1.

This work was supported in part by the Air Force Office of Scientific Research Grant no. F49620-80-C-001, by National Science Foundation Grant no. MCS-80-18294, and by a grant from the General Research Board of the University of Maryland. Computer time was provided in part by the Computer Science Center of the University of Maryland.

Tools can support slicing



Source tree:

```
+ (default package)
− edu.cmu.hcii.paint
  + Actions.java
  + EraserPaint.java
  + PaintCanvas.java
  + PaintObject.java
  + PaintObjectConstructor.java
  + PaintObjectConstructorListe
  − PaintWindow.java
    − PaintWindow$1.class
        PaintWindow$1()
        stateChanged()
    + PaintWindow$2.class
    + PaintWindow$3.class
    + PaintWindow.class
  − PencilPaint.java
    − PencilPaint.class
        PencilPaint()
        define()
        getBoundingBox()
        getEndX()
        getEndY()
        getStartX()
        getStartY()
        paint()
+ java.awt
+ java.awt.event
+ javax.swing
```

Code:

```
    private PaintObjectConstructor objectConstructor;

    private ChangeListener colorChange... = new ChangeListener() {

        public void stateChanged(ChangeEvent changeEvent) {

            objectConstructor.setColor(
                new Color(
                rSlider.getValue(),
                gSlider.getValue(),
                bSlider.getValue()));

            repaint();

        }
```

Popup:

```
Called Color() on <font color=#000000></b>
(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)
```

Q why did color = █?
A These events were responsible.

← event | → event | ← in method | → in method | ← in thread | → in thread | ⇧ block | collapse/expand | show threads

```
(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)
thread main-0 • • •   thread AWTEventQueue0-5 • • •   Color()   • • •   Color #19,941
start of program →
```

```
− AWTEventQueue0-5
  − PaintWindow$1 : stateChanged()
    + this = PaintWindow$1 #3,742
    + changeEvent = ChangeEvent
  + JSlider : fireStateChanged()
  + ModelListener : stateChanged()
  + DefaultBoundedRangeModel : fire
  + DefaultBoundedRangeModel : set
  + DefaultBoundedRangeModel : set
  + JSlider : setValueIsAdjusting()
  + TrackListener : mouseReleased()
  + Component : processMouseEven
  + JComponent : processMouseEve
```

Figure 1. Using the Whyline: (1) The developer demonstrates the behavior; (2) after the trace loads, the developer finds the output of interest by scrubbing the I/O history; (3) the developer clicks on the output and chooses a question; (4) the Whyline provides an answer, which the developer navigates (5) in order to understand the cause of the behavior (6).

Distinguished Paper

Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior

Amy J. Ko and Brad A. Myers
Human-Computer Interaction Institute
School of Computer Science, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213
{ajko, bam}@cs.cmu.edu

Abstract

When software developers want to understand the reason for a program's behavior, they must translate their questions about the behavior into a series of questions about code, speculating about the causes in the process. The Whyline is a new kind of debugging tool that avoids such speculation by instead enabling developers to select a question about program output from a set of *why did* and *why didn't* questions derived from the program's code and execution. The tool then finds one or more possible explanations for the output in question, using a combination of static and dynamic slicing, precise call graphs, and new algorithms for determining potential sources of values and explanations for why a line of code was not reached. Evaluations of the tool on one task showed that *novice* programmers *with* the Whyline were twice as fast as *expert* programmers *without* it. The tool has the potential to simplify debugging in many software development contexts.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids, tracing
H.5.2 [**User Interfaces**]: User centered design, interaction styles

General Terms

Reliability, Algorithms, Performance, Design, Human Factors.

1. INTRODUCTION

Software developers have long struggled with understanding the causes of software behavior. And yet, despite decades of knowing that program understanding and debugging are some of the most challenging and time consuming aspects of software development, little has changed in how people work: these tasks still represent up to 70% of the time required to ship a software product [17].

A simple problem underlies this statistic: once a person sees an inappropriate behavior, they must then translate their questions about the *behavior* into a series of queries about the program's *code*. In doing this translation, developers basically have to guess about what code is responsible [10]. This is worsened by the fact that bugs often manifest themselves in strange and unpredictable ways: a typo in a crucial conditional can dramatically alter program behavior. Even for experienced developers, speculation about the relationship between the *symptoms* of a problem and their *cause* is a serious issue. In our investigations, developers' initial guesses were wrong almost 90% of the time [7,8].

Unfortunately, today's debugging and program understanding tools do not help with this part of the task. Breakpoint debuggers require people to choose a line of code. Slicing tools require a choice of variable [2]. Querying tools require a person to write an executable expression about data [11]. As a result, all of these tools are subject to a 'garbage-in garbage-out' limitation: if a developer's choice of code is irrelevant to the cause, the tool's answer will be similarly irrelevant. Worse yet, *none* of today's tools allow developers to ask *why not* questions about things that did not happen; such questions are often the majority of developers' questions [10]. Of course, lots of things do not happen in a program, but developers tend only to ask about behaviors that a program is *designed* to do.

In this paper, we present a new kind of program understanding and debugging tool called a *Whyline*, which overcomes these limitations. The idea is simple: rather than requiring people to translate their questions to code queries, the Whyline allows developers to choose a *why did* or *why didn't* question about program output and then the Whyline generates an answer to the question using a variety of program analyses. This avoids the problems noted above because developers are much better at reasoning about program output, since unlike the execution of code, it is observable. Furthermore, in many cases, developers themselves define correctness in *terms* of the output.

This work follows earlier prototypes. The Alice Whyline [8] supported a similar interaction technique, but for an extremely simple language with little need for procedures and a rigid definition of output (in a lab study, the Whyline for Alice decreased debugging time by a factor of 8). The Crystal framework [14], which supported questions in end-user applications, applied the same ideas, but limited the scope mostly to questions about commands and events that appear in an application's undo stack (in lab studies of Crystal, participants were able to complete 30% more tasks, 20% faster).

These successes inspired us to extend these ideas to an implementation for *Java*, which removes many of the limitations of our earlier work. We contribute (1) algorithms for deriving questions from code that are efficient and output-relevant, (2) algorithms for answering questions that provide near immediate feedback, and (3) a visualization of answers that is compact and simple to navigate. We achieve all of this with no limitations on the target program, other than that it uses standard Java I/O mechanisms and that the program does not run too long (given our trace-based approach).

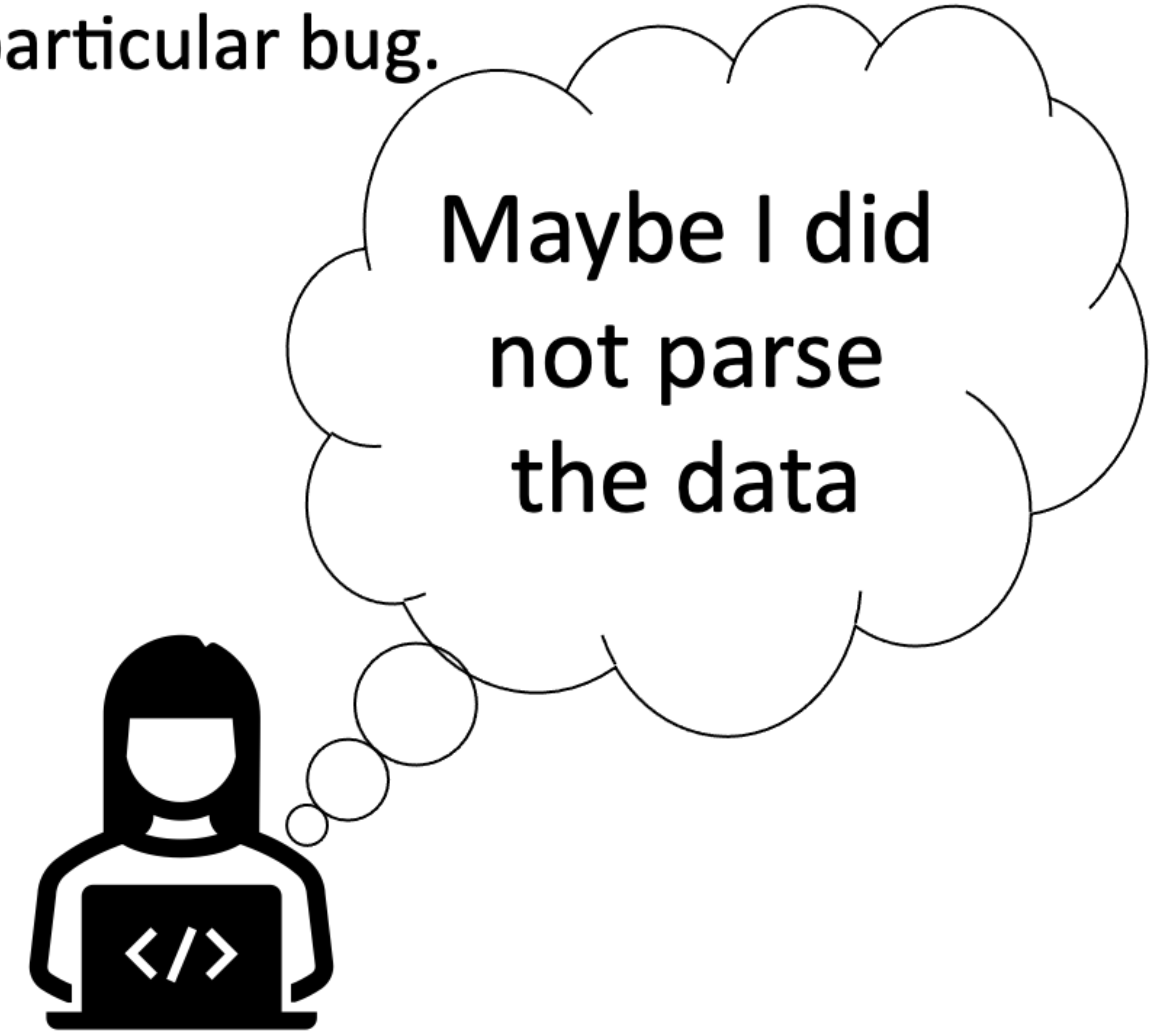
Limitations of Slicing

- May not work well for
  - Questions about what **did not** happen
  - Interactions involving API behavior
  - Long running operations

Why didn't this happen? (3)
How do I debug this bug in this environment? (3)
In what circumstances does this bug occur? (3)

Debugging hypotheses

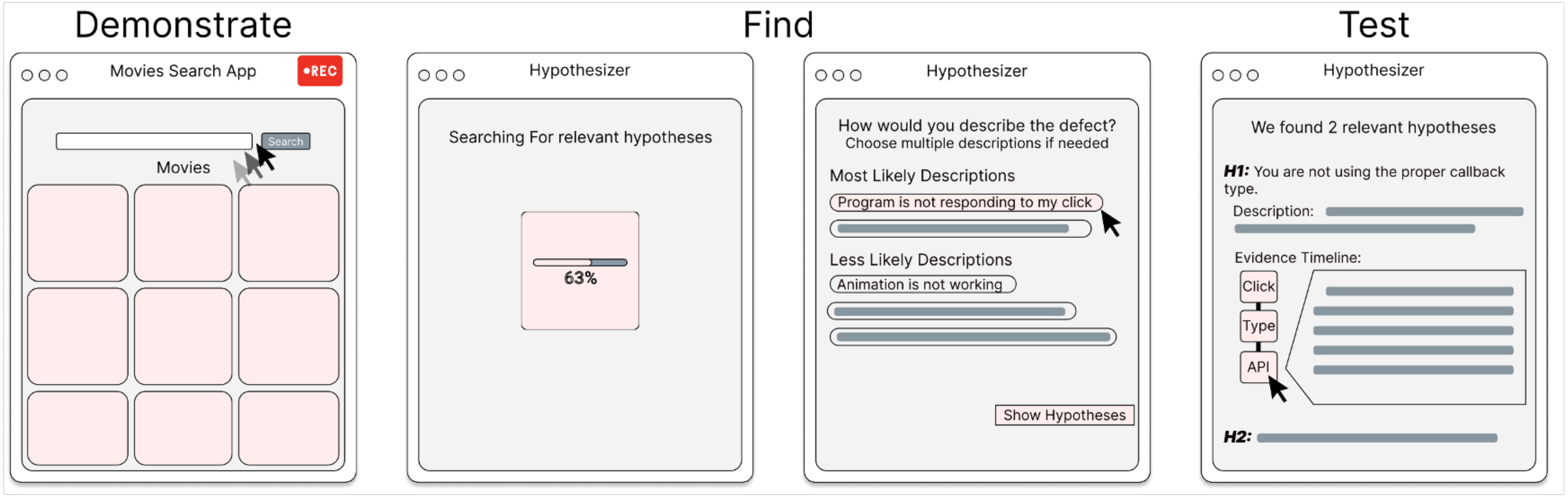
- a *hypothesis* is an educated guess about what might be causing a particular bug.



Work to test the hypothesis

Read related code 📄📄
Look at relevant online resources 🔍
Debug specific of line of code 👨‍💻

Hypothesis-Based Debugging: Hypothesizer

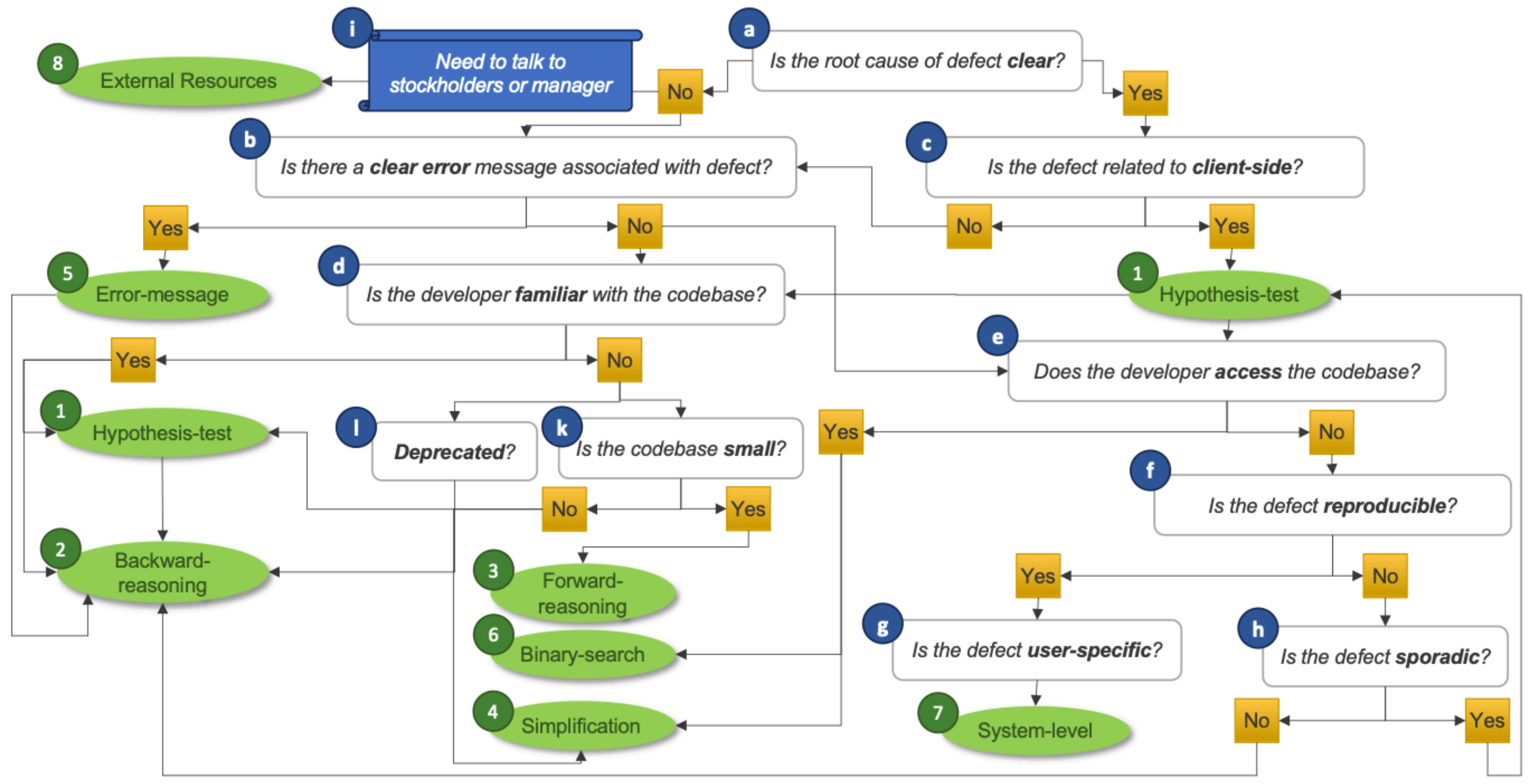


Abdulaziz Alaboudi and Thomas D. Latoza. 2023. Hypothesizer: A Hypothesis-Based Debugger to Find and Test Debugging Hypotheses. In Symposium on User Interface Software and Technology, 1–14. https://doi.org/10.1145/3586183.3606781

Debugging strategies

- Developers follow debugging strategies to debug a defect.
- Developers choose different strategies depending on context factors.

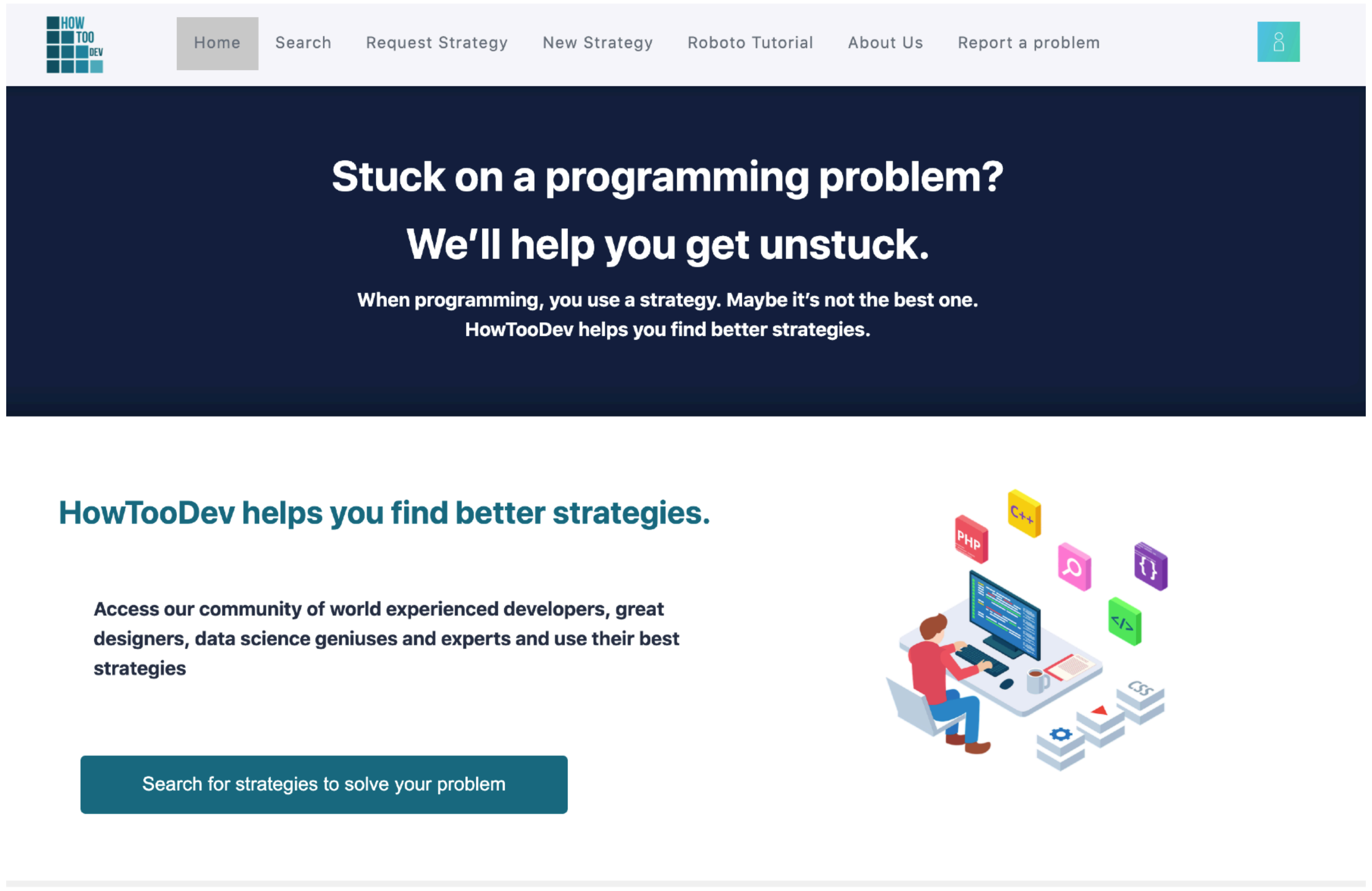
1. Hypothesis-test	Developing hypotheses about the cause of the defect and testing these hypotheses by gathering evidence in the code or runtime behavior.
2. Backward-reasoning	Identify solutions or diagnoses by tracing the error's manifestations back through the code execution path to uncover the underlying possible causes.
3. Forward-reasoning	Starting with an initial state or known facts. move forward logically from the initial state, applying rules, operations, or statements to generate new facts or states, systematically examining each step towards the goal or the discovery of a defect root.
4. Simplification	Breaking down the problem into smaller, more manageable parts, removing unnecessary details, and focusing on the core aspects that are crucial for finding a solution.
5. Error-message	Understanding the content and meaning of error messages, which often include error codes, descriptions, and context about where and why the error occurred. followed by reading documentation, online resources, forums, and knowledge bases to look up error.
6. Binary-search	Repeatedly dividing the codebase or input space into smaller sections and testing each section to isolate the problematic area.



a Is the root cause of defect clear?
Yes
No
i Need to talk to stockholders or manager
8 External Resources
b Is there a clear error message associated with defect?
c Is the defect related to client-side?
Yes
No
No
Yes
5 Error-message
d Is the developer familiar with the codebase?
1 Hypothesis-test
e Does the developer access the codebase?
Yes
No
1 Hypothesis-test
l Deprecated?
k Is the codebase small?
Yes
No
No
Yes
f Is the defect reproducible?
2 Backward-reasoning
3 Forward-reasoning
6 Binary-search
4 Simplification
Yes
No
g Is the defect user-specific?
h Is the defect sporadic?
7 System-level
No
Yes

Explicit debugging strategies

- Can teach developers the best strategy for a specific defect.



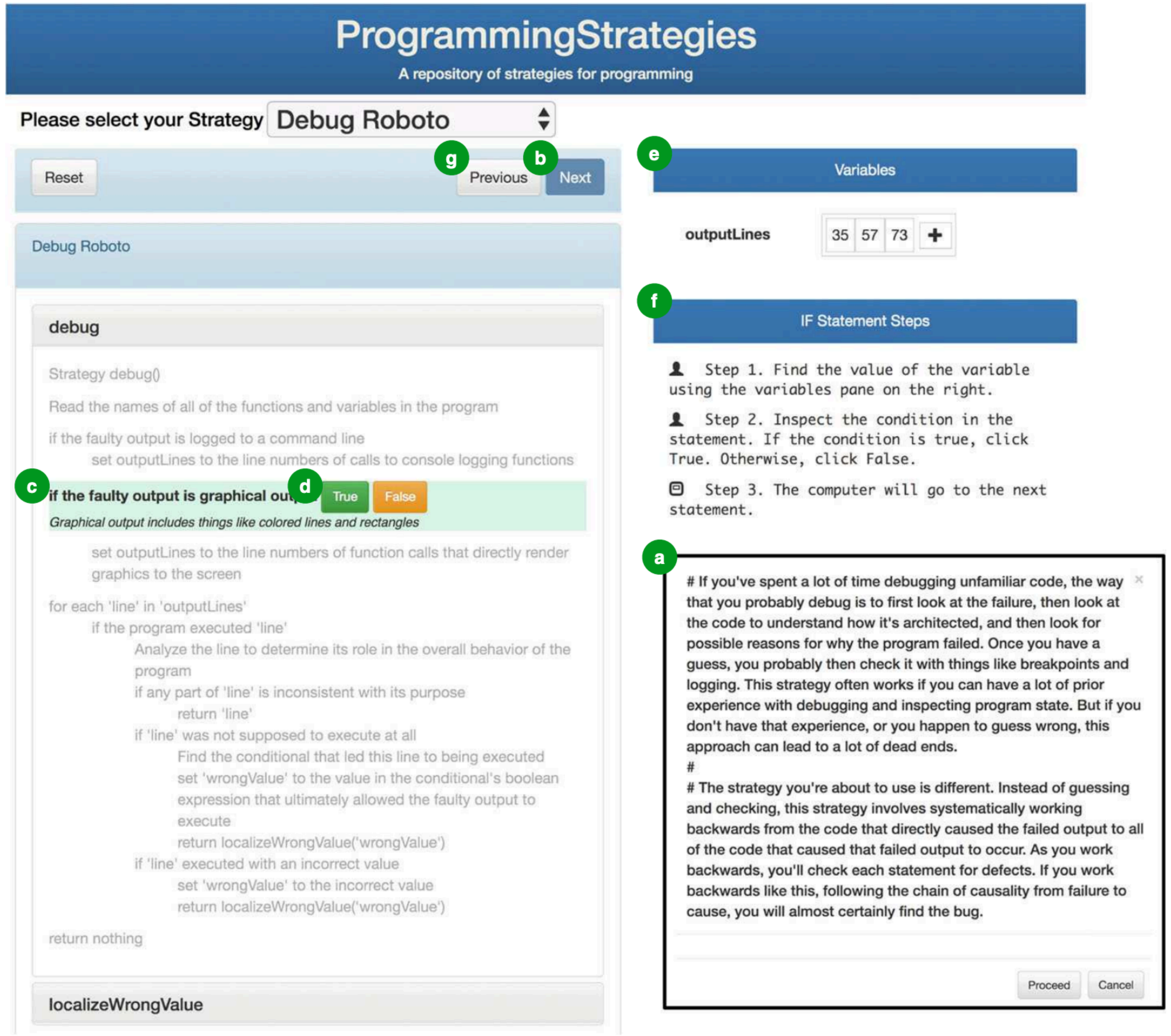
Stuck on a programming problem?

We'll help you get unstuck.

When programming, you use a strategy. Maybe it's not the best one. HowTooDev helps you find better strategies.

HowTooDev helps you find better strategies.

Access our community of world experienced developers, great designers, data science geniuses and experts and use their best strategies



Please select your Strategy: Debug Roboto

Debug Roboto

debug

```
Strategy debug()
Read the names of all of the functions and variables in the program
if the faulty output is logged to a command line
    set outputLines to the line numbers of calls to console logging functions
if the faulty output is graphical output
Graphical output includes things like colored lines and rectangles
    set outputLines to the line numbers of function calls that directly render
    graphics to the screen
for each 'line' in 'outputLines'
    if the program executed 'line'
        Analyze the line to determine its role in the overall behavior of the
        program
        if any part of 'line' is inconsistent with its purpose
            return 'line'
        if 'line' was not supposed to execute at all
            Find the conditional that led this line to being executed
            set 'wrongValue' to the value in the conditional's boolean
            expression that ultimately allowed the faulty output to
            execute
            return localizeWrongValue('wrongValue')
        if 'line' executed with an incorrect value
            set 'wrongValue' to the incorrect value
            return localizeWrongValue('wrongValue')
return nothing
```

localizeWrongValue

outputLines	35	57	73

Step 1. Find the value of the variable using the variables pane on the right.

Step 2. Inspect the condition in the statement. If the condition is true, click True. Otherwise, click False.

Step 3. The computer will go to the next statement.

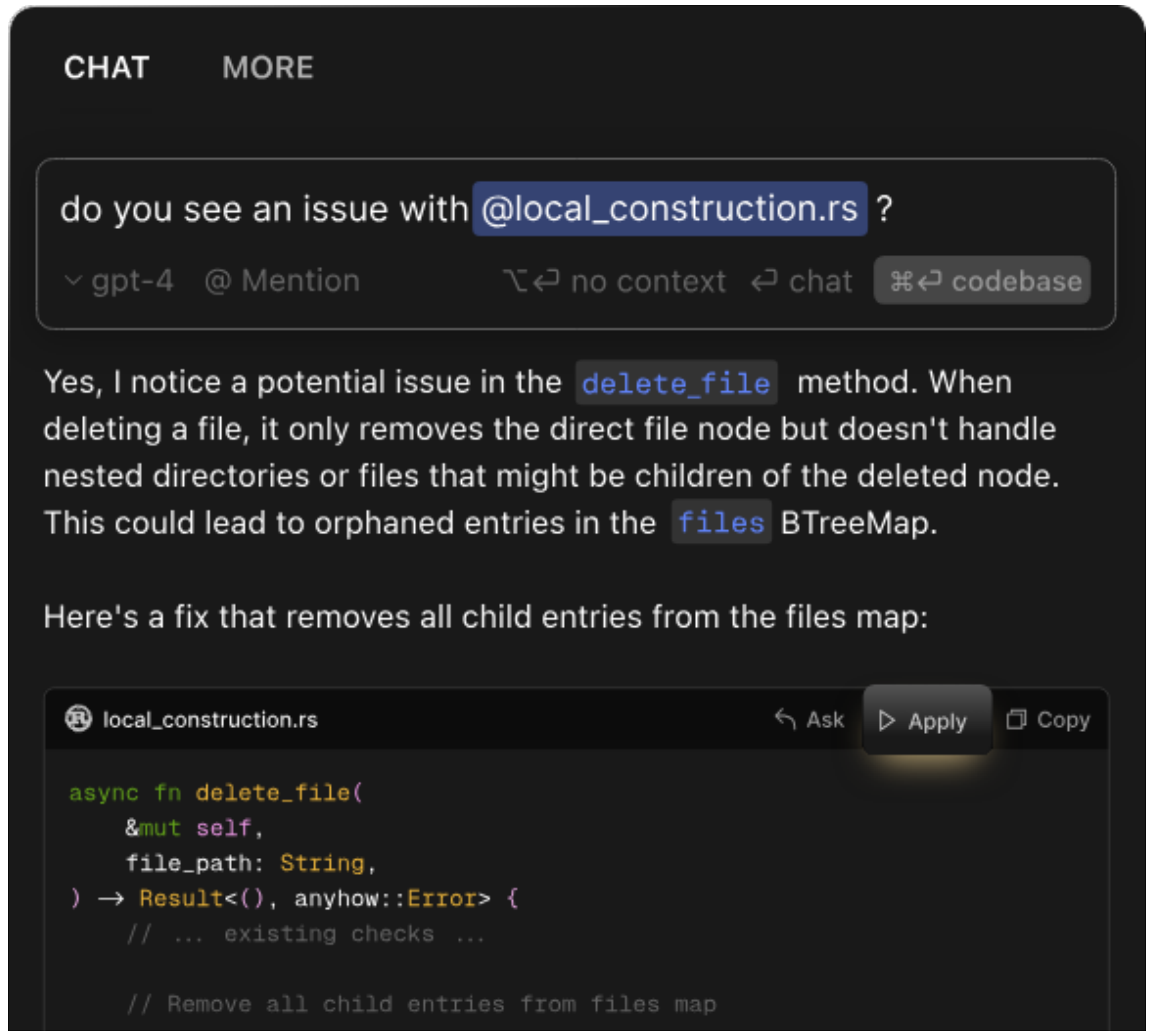
If you've spent a lot of time debugging unfamiliar code, the way that you probably debug is to first look at the failure, then look at the code to understand how it's architected, and then look for possible reasons for why the program failed. Once you have a guess, you probably then check it with things like breakpoints and logging. This strategy often works if you can have a lot of prior experience with debugging and inspecting program state. But if you don't have that experience, or you happen to guess wrong, this approach can lead to a lot of dead ends.
#
The strategy you're about to use is different. Instead of guessing and checking, this strategy involves systematically working backwards from the code that directly caused the failed output to all of the code that caused that failed output to occur. As you work backwards, you'll check each statement for defects. If you work backwards like this, following the chain of causality from failure to cause, you will almost certainly find the bug.

M. Arab, J. Liang, Y. Yoo, A. J. Ko and T. D. LaToza. (2021). "HowToo: A Platform for Sharing, Finding, and Using Programming Strategies," *Symposium on Visual Languages and Human-Centric Computing,* 1-9, doi: 10.1109/VL/HCC51201.2021.9576337.

LaToza, T.D., Arab, M., Loksa, D. *et al.* Explicit programming strategies. *Empir Software Eng* **25**, 2416–2449 (2020). https://doi.org/10.1007/s10664-020-09810-1

Debugging in the Age of LLMs

- LLMs can already help debug
    - Can help diagnose and fix code involving incorrect API interactions
    - If some of these are hard defects, could already be a big win!
- More challenging for defects that
    - less connected to API interactions
    - require collecting information scattered across files or through debugging tools
    - require dynamic execution information rather than static code text



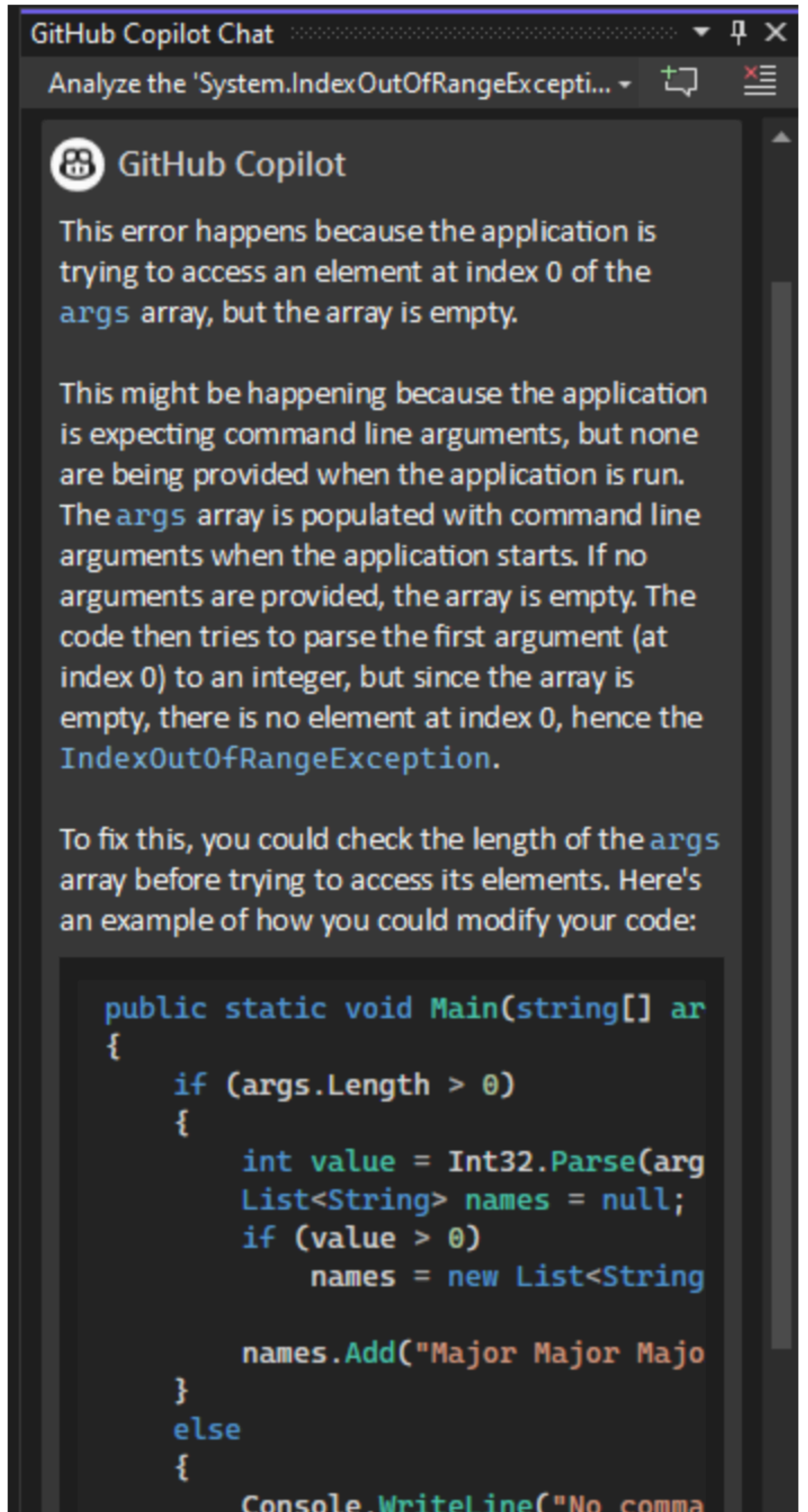
do you see an issue with @local_construction.rs ?

Yes, I notice a potential issue in the `delete_file` method. When deleting a file, it only removes the direct file node but doesn't handle nested directories or files that might be children of the deleted node. This could lead to orphaned entries in the `files` BTreeMap.

Here's a fix that removes all child entries from the files map:

```
async fn delete_file(
    &mut self,
    file_path: String,
) → Result<(), anyhow::Error> {
    // ... existing checks ...

    // Remove all child entries from files map
```



This error happens because the application is trying to access an element at index 0 of the `args` array, but the array is empty.

This might be happening because the application is expecting command line arguments, but none are being provided when the application is run. The `args` array is populated with command line arguments when the application starts. If no arguments are provided, the array is empty. The code then tries to parse the first argument (at index 0) to an integer, but since the array is empty, there is no element at index 0, hence the `IndexOutOfRangeException`.

To fix this, you could check the length of the `args` array before trying to access its elements. Here's an example of how you could modify your code:

```
public static void Main(string[] ar
{
    if (args.Length > 0)
    {
        int value = Int32.Parse(arg
        List<String> names = null;
        if (value > 0)
            names = new List<String

        names.Add("Major Major Majo
    }
    else
    {
        Console.WriteLine("No comma
```

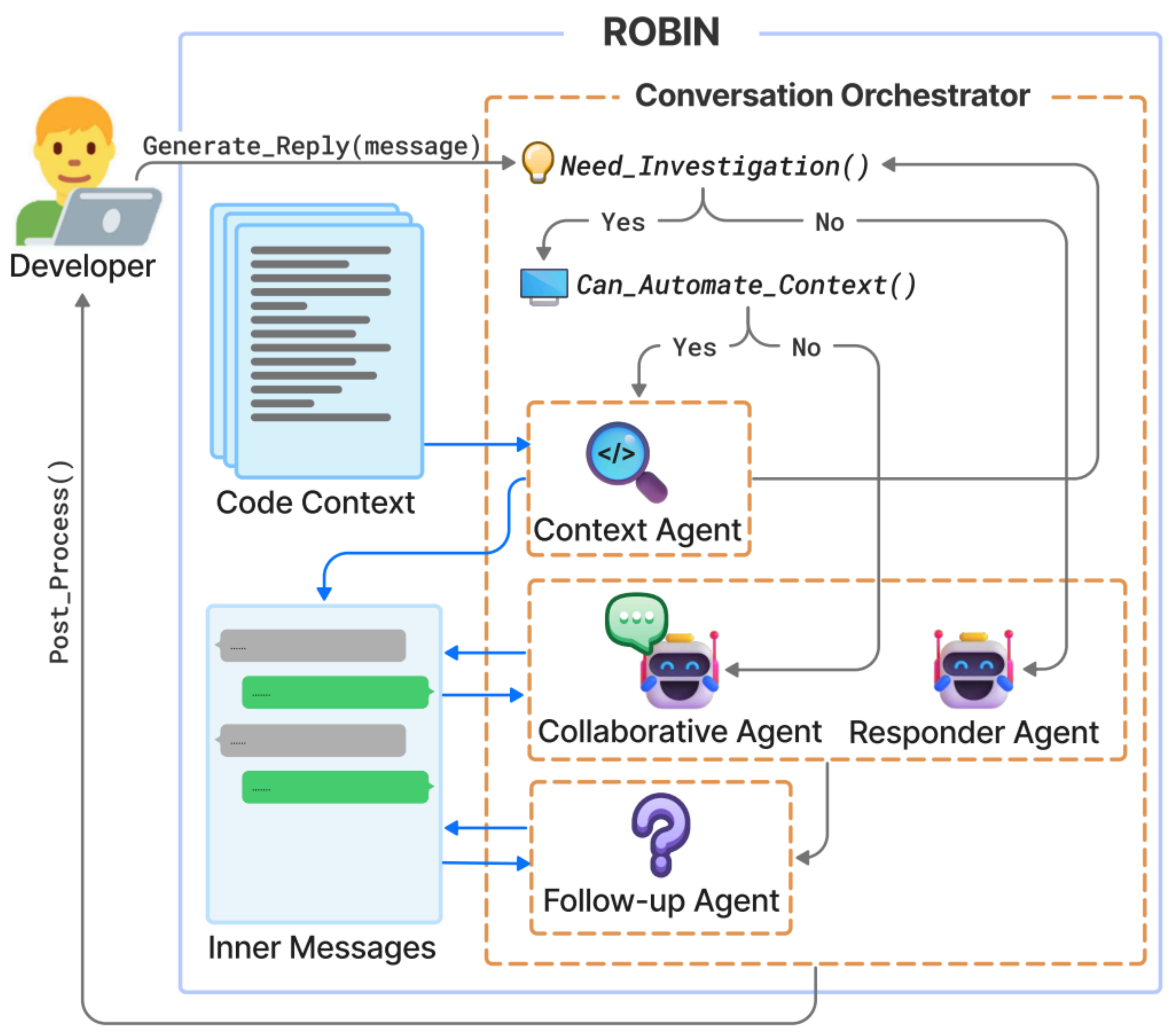
Supporting debugging with LLMs

- Slicing: interpret information along a slice
- Fault localization: generate explanation & fix
- Hypothesis testing: generate hypothesis & gather evidence and work together to test
- Strategies: script IDE interactions to gather information

```
if (error) {
  console.log("Error:" + error);
  process.exit(1);
}
```

Supporting explicit debugging strategies

- Programming strategies can help direct agents LLMs.
- Offloads some of the boilerplate of gathering info through strategies to LLM



Yasharth Bajpai, Bhavya Chopra, Param Biyani, Cagri Aslan, Sumit Gulwani, Dustin Coleman, Chris Parnin, Arjun Radhakrishna, Gustavo Soares. Let's Fix this Together: Conversational Debugging with GitHub Copilot. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) | August 2024 **Best Research Paper Award**

Theories of Program Comprehension in the Age of LLMs

- How can developers still understand the code being generated?

  ⇒ Theories of Information Needs in Programming

- To what extent do developers really have to understand the code being written?

  ⇒ Theories of Information Hiding

- How do developers figure out what's wrong when it doesn't work?

  ⇒ Theories of Debugging

Where do you design your tool to intervene?

- LLM helps to
  - Translate a requirements doc into code
  - Generate a new design doc from code
  - Fix issue in issue tracker
  - Diagnose & fix a defect
  - Answer a complex question
  - Execute strategy
  - Generate a few lines of code
  - Produce a simple fact about code

High-level task

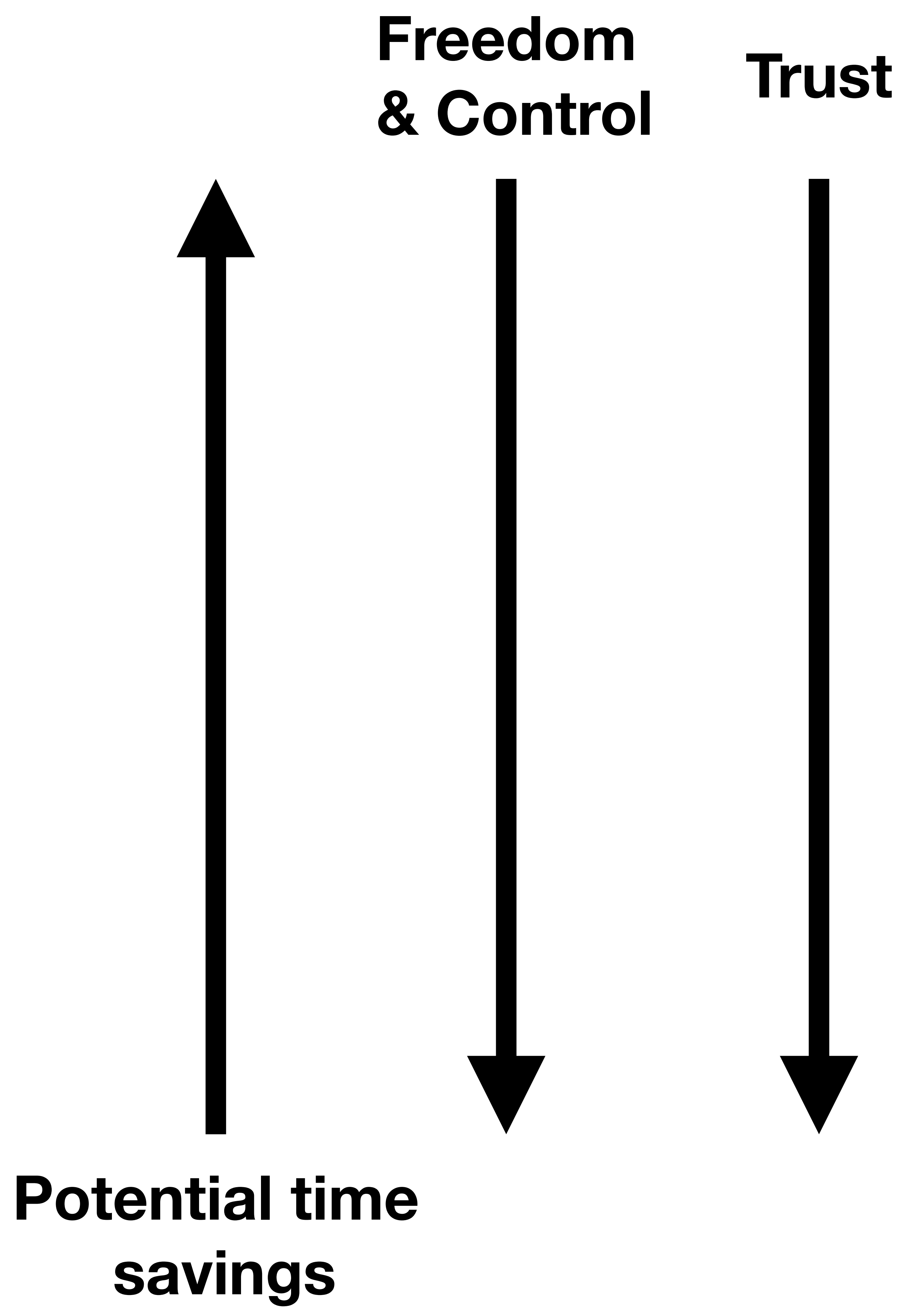
Low level task

Spec-driven development

Auto coders

Agentic programming

VS Code



Charting a future of LLMs & program comprehension w/ Theories

- Traditional cognitivist theories of developer behavior
  - Information needs, strategies, schemas,
- Theories of why tools help

- Build more theories by explicitly state what we believe, where we learned this
- Identify conflicting beliefs, use studies to test
- Create new tools informed by problems & activities from theories

Theories of Program Comprehension in the Age of LLMs

Theories of Information Needs in Programming

- Developers ask **questions**

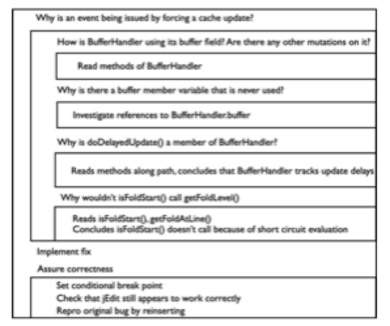
What does this do when input is null?
What part of this is being done client side and what part server side?

- Questions are **task-specific**

debugging refactoring testing testing

- Answering questions raises **more questions**.

- Tool which successfully supports the questions a developer asks increases their productivity



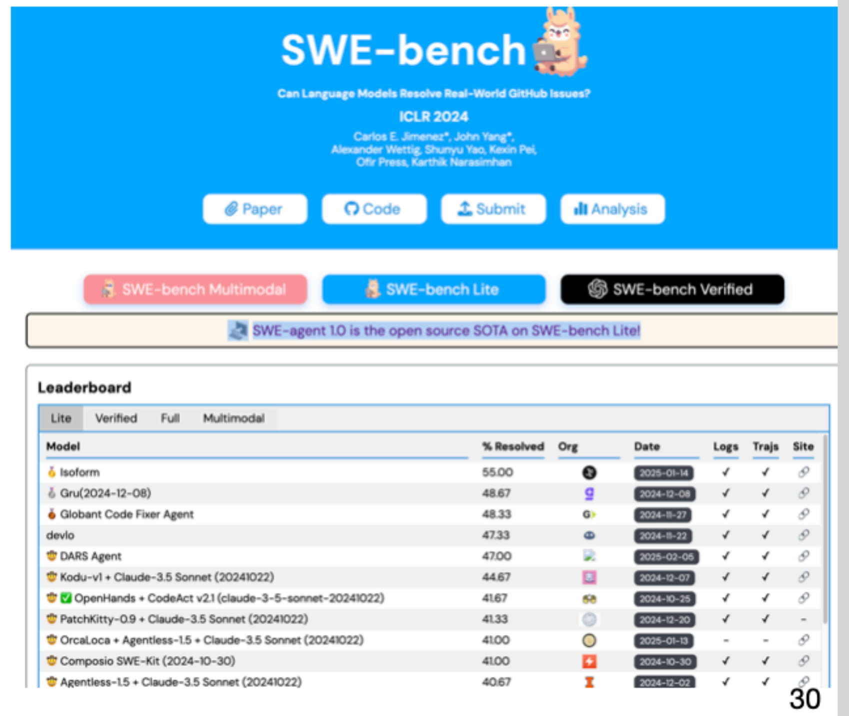
Why is an event being issued by forcing a cache update?
How is BufferHandler using its buffer field? Are there any other mutations on it?
Read methods of BufferHandler
Why is there a buffer member variable that is never used?
Investigate references to BufferHandler.buffer
Why is doDelayedUpdate() a member of BufferHandler?
Reads methods along path, concludes that BufferHandler tracks update delays
Why wouldn't isFoldStart() call getFoldLevel()
Reads isFoldStart(), getFoldAtLine()
Concludes isFoldStart() doesn't call because of short circuit evaluation
Implement fix
Assure correctness
Set conditional break point
Check that jEdit still appears to work correctly
Repro original bug by reinserting

16

Benchmarking Program Comprehension with LLMs

Slide Subtitle

- Existing benchmarks examine full automation of SE (e.g., success in generating a patch)
  - Many tasks will still require developer involvement.
  - Want to have benchmarks that measure benefit to program comprehension of an LLM assistant
  - Can use hard to answer questions as benchmark



SWE-bench

Can Language Models Resolve Real-World GitHub Issues?
ICLR 2024
Carlos E. Jimenez*, John Yang*, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, Karthik Narasimhan

SWE-agent 1.0 is the open source SOTA on SWE-bench Lite!

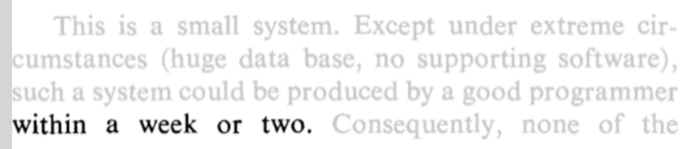
Leaderboard

Model	% Resolved	Org	Date	Logs	Trajs	Site
Isoform	55.00		2025-01-14	✓	✓	🔗
Gru(2024-12-08)	48.67		2024-12-08	✓	✓	🔗
Globant Code Fixer Agent	48.33		2024-11-27	✓	✓	🔗
devlo	47.33		2024-11-22	✓	✓	🔗
DARS Agent	47.00		2025-02-05	✓	✓	🔗
Kodu-v1 + Claude-3.5 Sonnet (20241022)	44.67		2024-12-07	✓	✓	🔗
✅ OpenHands + CodeAct v2.1 (claude-3-5-sonnet-20241022)	41.67		2024-10-25	✓	✓	🔗
PatchKitty-0.9 + Claude-3.5 Sonnet (20241022)	41.33		2024-12-20	✓	✓	-
OrcaLoca + Agentless-1.5 + Claude-3.5 Sonnet (20241022)	41.00		2025-01-13	-	-	🔗
Composio SWE-Kit (2024-10-30)	41.00		2024-10-30	✓	✓	🔗
Agentless-1.5 + Claude-3.5 Sonnet (20241022)	40.67		2024-12-02	✓	✓	🔗

30

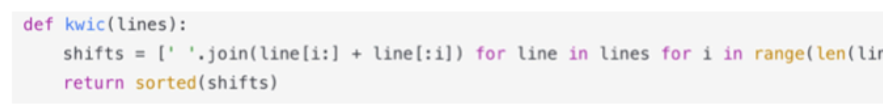
Powerful abstractions help build more quickly

- Parnas' **Key Words in Context** Problem, used to illustrate 1972 paper on information hiding
  - The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWXC index system outputs a listing of all circular shifts of all lines in alphabetical order.



This is a small system. Except under extreme circumstances (huge data base, no supporting software), such a system could be produced by a good programmer within a week or two. Consequently, none of the

1972



```
def kwic(lines):
    shifts = [' '.join(line[i:] + line[:i]) for line in lines for i in range(len(lin
    return sorted(shifts)
```

2025

D. L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623

Theories of Abstraction in the Age of LLMs

- Small teams of expert software engineers build the hard building blocks that require deep system expertise, exposed as framework and libraries
- Everyone else vibe codes with LLMs to build small ephemeral apps on top of the underlying capabilities of LLMs

41

Theories of Debugging

Slide Subtitle

- Fault localization: debugging is the process of finding the line with the defect
- Slicing: navigate forwards or backwards across control & data dependencies to locate the defective line
- Strategies: follow a strategy to investigate the code in a systematic way
- Hypothesis testing: use intuition to form hypothesis and sytematically gather evidence to accept or reject

47

Where do you design your tool to intervene?

Slide Subtitle

- LLM helps to
  - Translate a requirements doc into code
  - Generate a new design doc from code
  - Fix issue in issue tracker
  - Diagnose & fix a defect
  - Answer a complex question
  - Execute strategy
  - Generate a few lines of code
  - Produce a simple fact about code

-level task

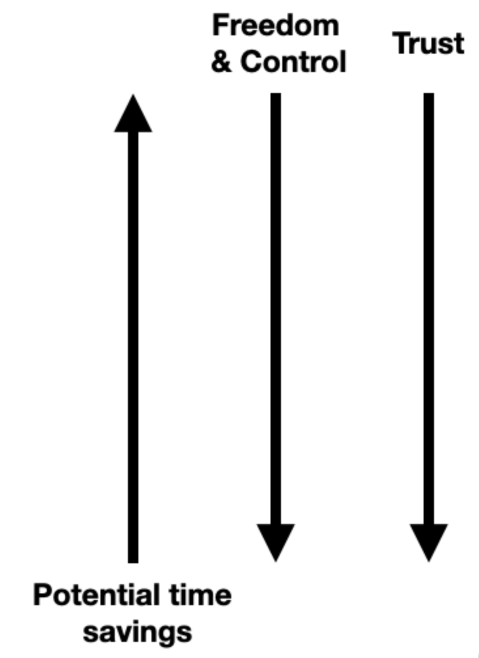
w level task

Spec-driven development

Auto coders

Agentic programming

VS Code


Freedom & Control
Trust
Potential time savings