Debugging Part 1

CS 691 / SWE 699

Fall 2025

GEORGE MASON UNIVERSITY

Logistics

- Reflection 2 & Lecture 5 reading questions due today at 4:30pm
- Lecture 6 reading questions for next week due 10/2 at 4:30pm
- Lecture 5 activity (in class today), due by 9/26 at 4:30pm
- No HW assignment assigned today (next one will be assigned on 10/2)

Today

- Discussion: Experiences from Lecture 4
- Discussion: Reading questions for Lecture 5
- Lecture
    - Debugging
- In-Class Activity

Discussion: Experiences from Lecture 4 Activity

- How did your tool try to address code quality issues?
- How did it support the developer in trying to see these issues
- What types of issues was it able to find?
- What was hard about using Cursor to try to build this?

Discussion: Reading questions for Lecture 5

- What questions did you have from readings from Lecture 5
  - Discuss questions & possible answers in group of 3 or 4
  - Come back with 1 question you want to discuss w/ whole class

Debugging

Overview

- How do human developers debug
  - Wide variety of tools & techniques
  - Challenges that impede progress, lead to necessity to choose other strategies
- How do LLMs learn how to debug from seeing what humans do
  - Example of LLM debugging
  - Examples of data LLMs can learn from

Bug Investigation Example

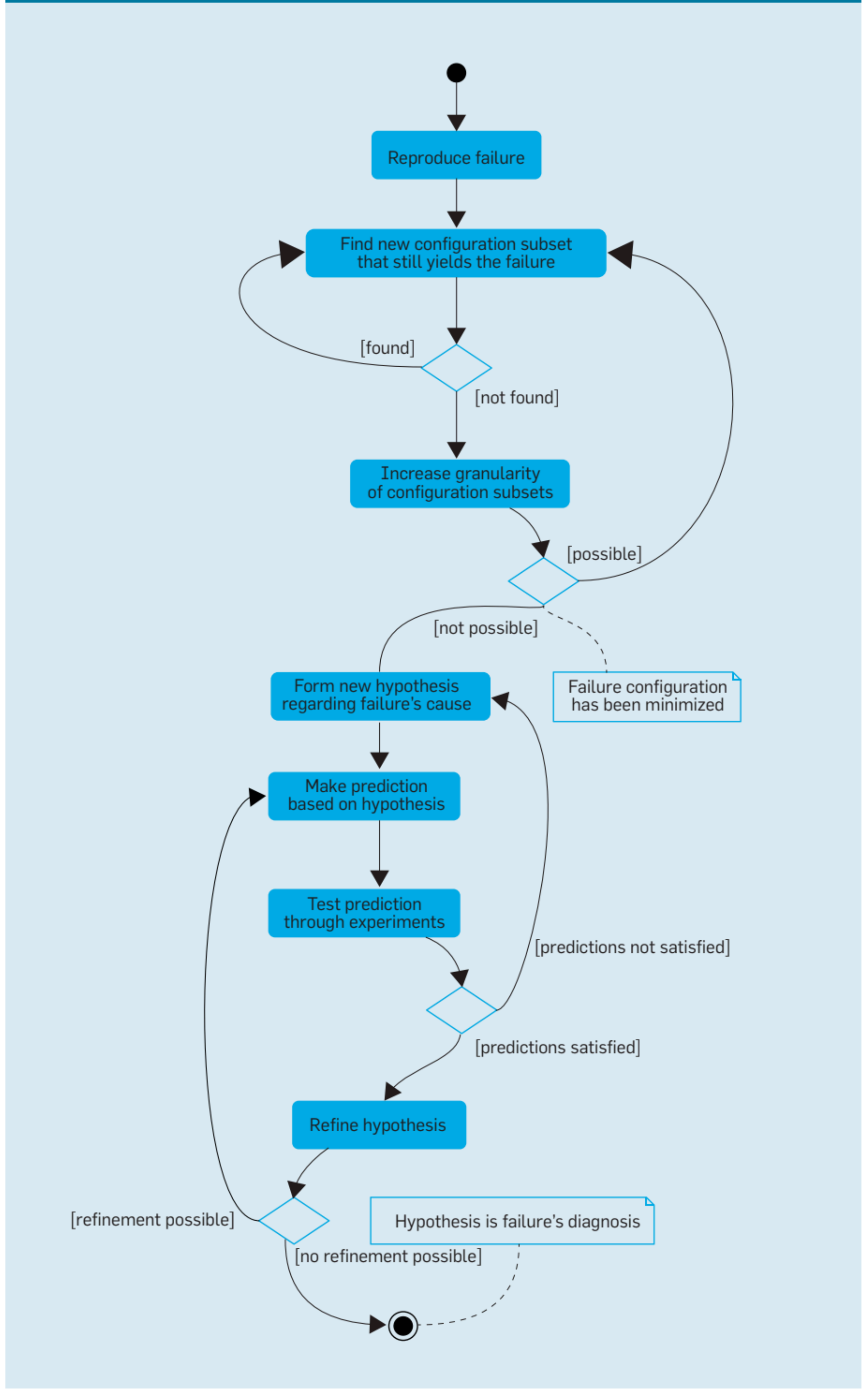
1. SDE assigned bug through Product Studio
2. Reproduces error
   - Browser hits error message (500 internal error)
3. Attach VS debugger
   - Browse to page again, hit null reference exception
4. Hypothesize from call stack which function might be responsible
5. Switch to emacs to browse through code
   - Ctags.exe works, VS code navigation broken
6. Switch back to debugger to change values & experiment
7. Make change, recompile, check, doesn't work
8. Navigates slice, wrong values came from object1, from object2, from object3 w/ mutexes
   - In complicated code doesn't understand
9. Walks to developer2's office and asks where data comes from
   - Developer2 working on high profile feature in area
10. Tries to make change, still doesn't work
11. Walks back to developer2, realize related to developer3 feature, developer3 at lunch
12. After lunch, developer1 and developer2 walk to developer3's office, change design to work with new feature
13. Bug passed from developer1 to developer3 to change feature

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). Association for Computing Machinery, New York, NY, USA, 492–501. https://doi.org/10.1145/1134285.1134355

Debugging process

- Reproduce the problem
- **Find cause of defect (fault localization)**
- Investigate fix
- Implement fix
- Test fix

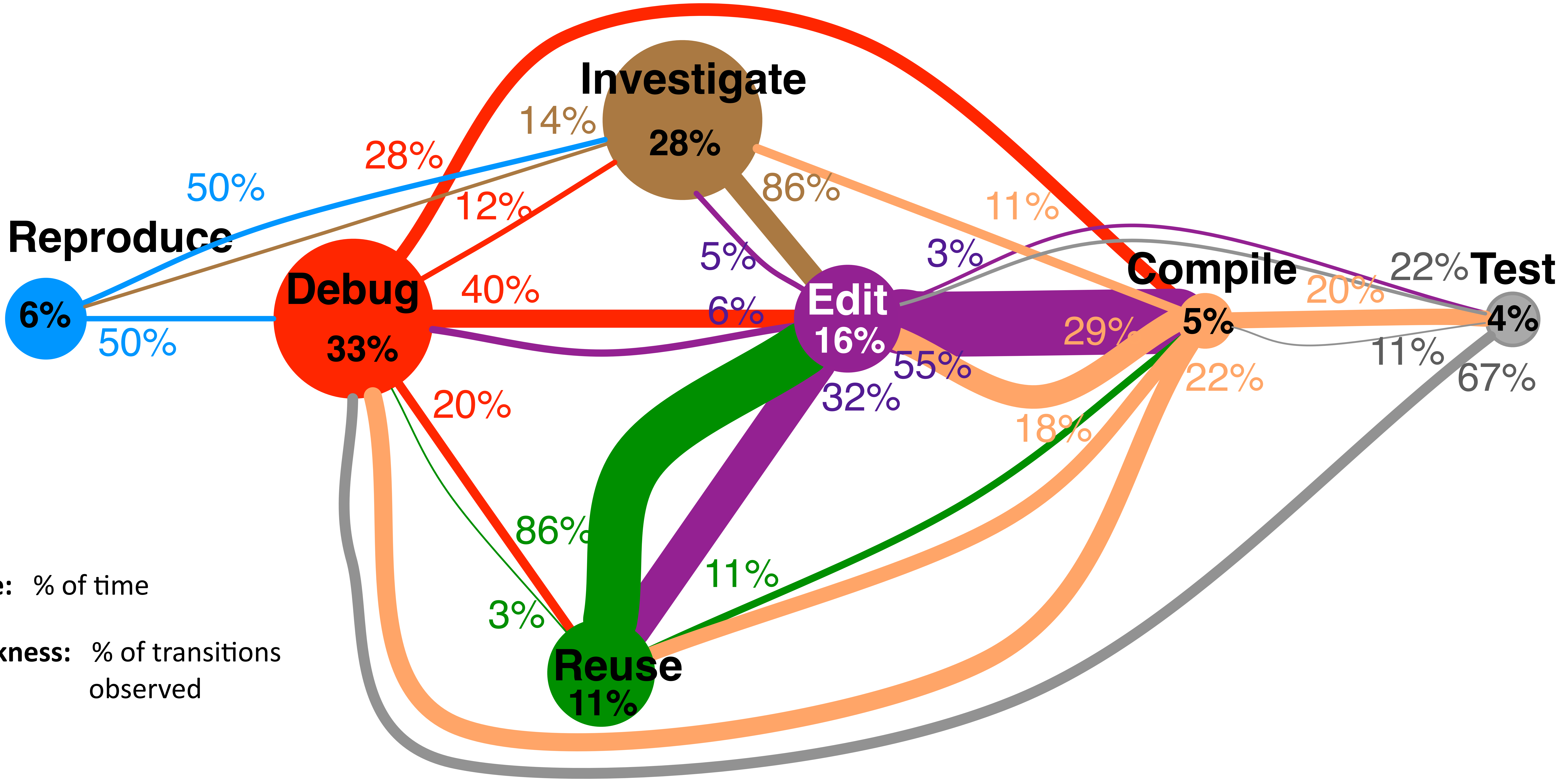
Figure 1. A process for systematic debugging.



Reproduce failure
Find new configuration subset that still yields the failure
[found]
[not found]
Increase granularity of configuration subsets
[possible]
[not possible]
Failure configuration has been minimized
Form new hypothesis regarding failure's cause
Make prediction based on hypothesis
Test prediction through experiments
[predictions not satisfied]
[predictions satisfied]
Refine hypothesis
[refinement possible]
Hypothesis is failure's diagnosis
[no refinement possible]

Edit / Debug Cycle

For tasks in code in your own codebase that you haven't seen recently



Circle size: % of time

Edge thickness: % of transitions observed

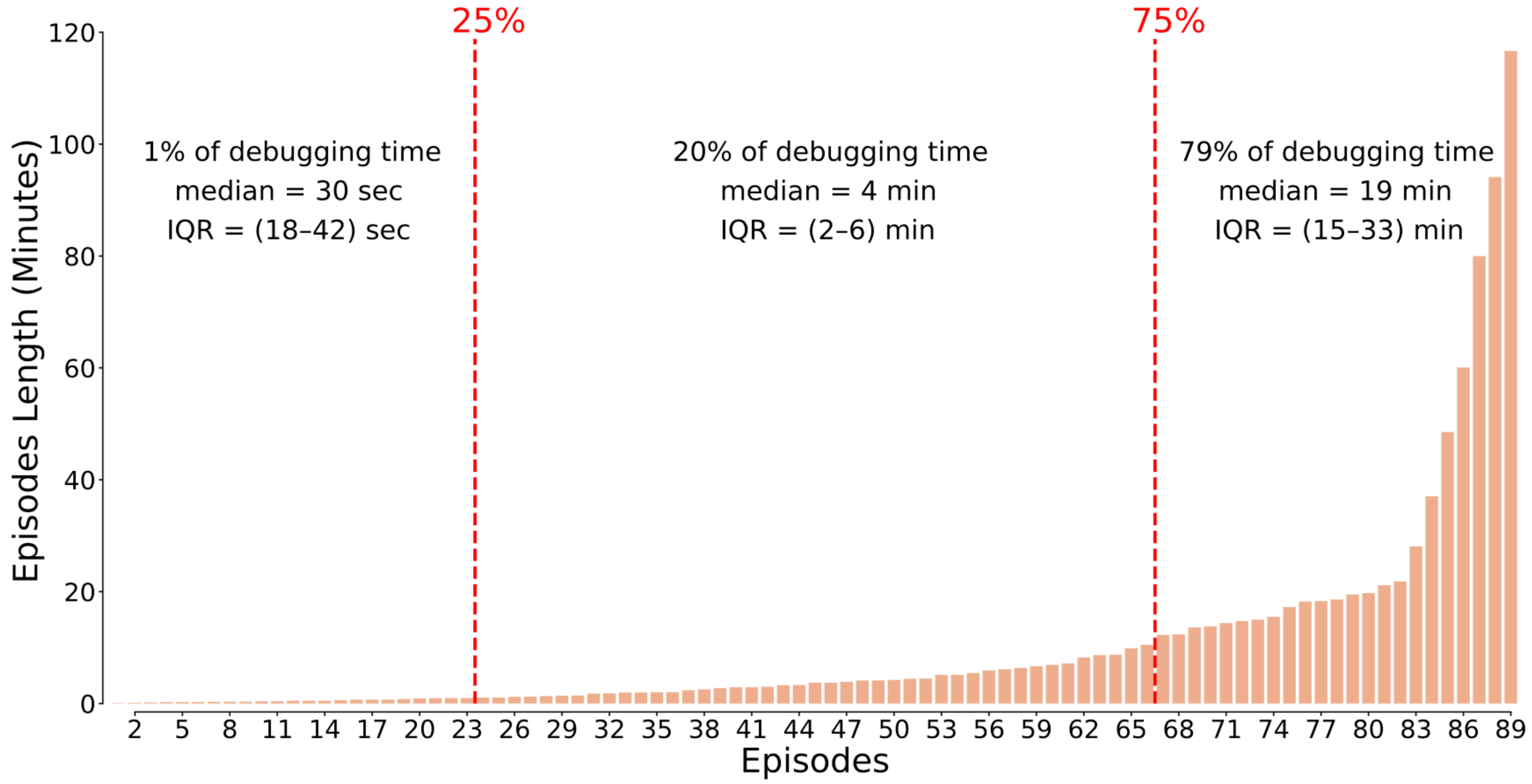


Fig. 2 Debugging episode length, from shortest to longest

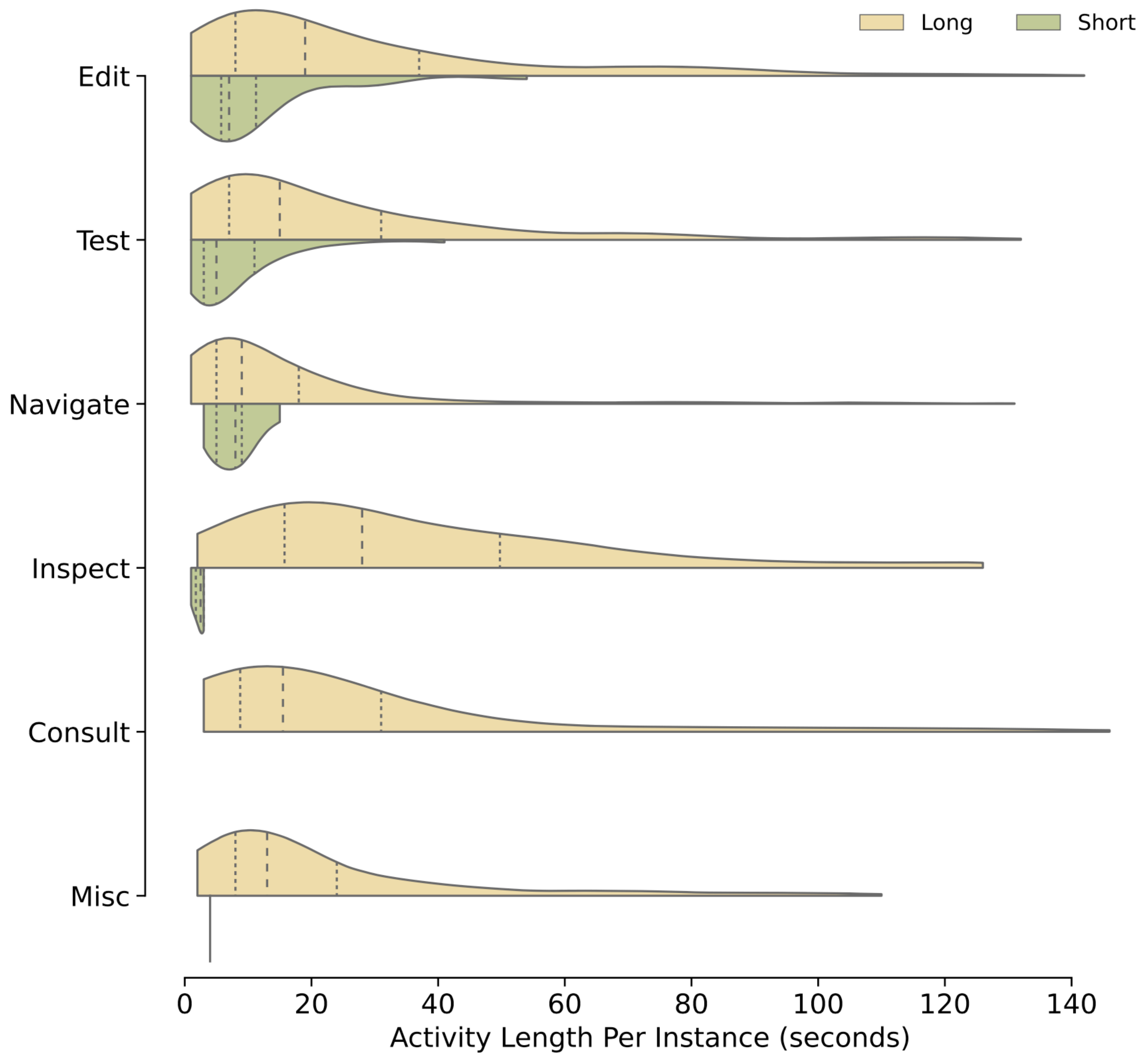


Fig. 3 The distribution of the time developers spent on each activity instance in long and short debugging episodes

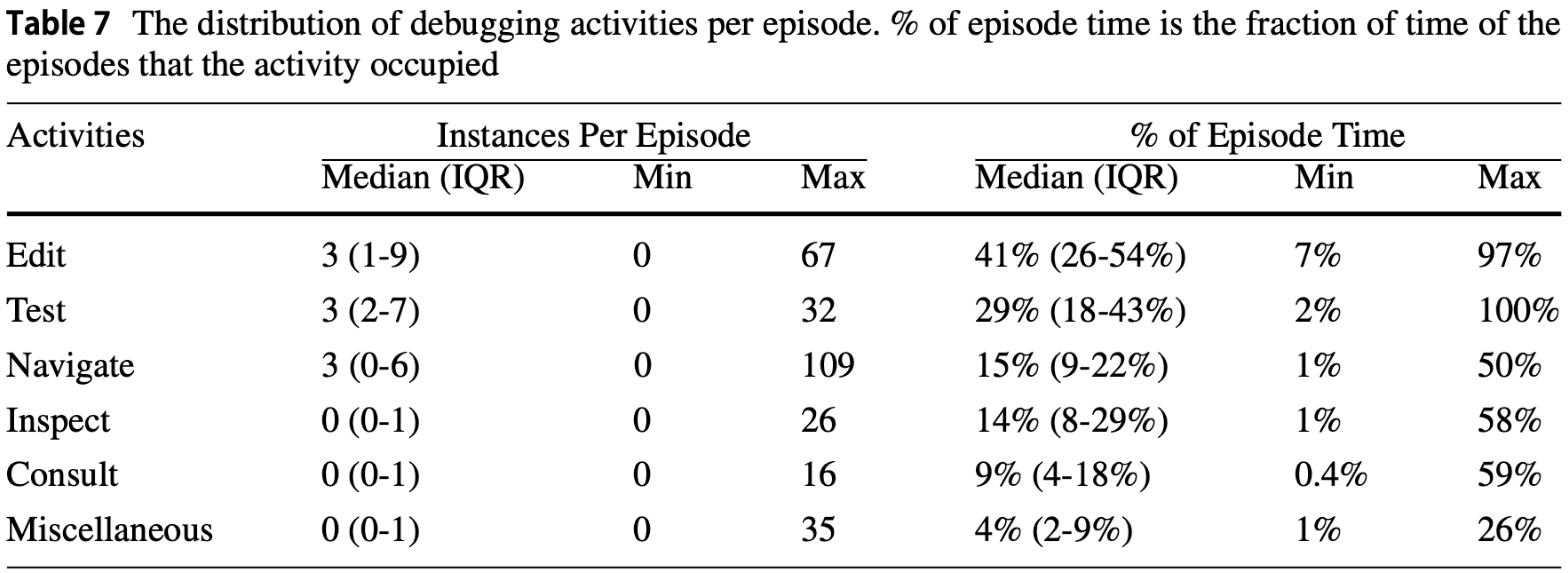
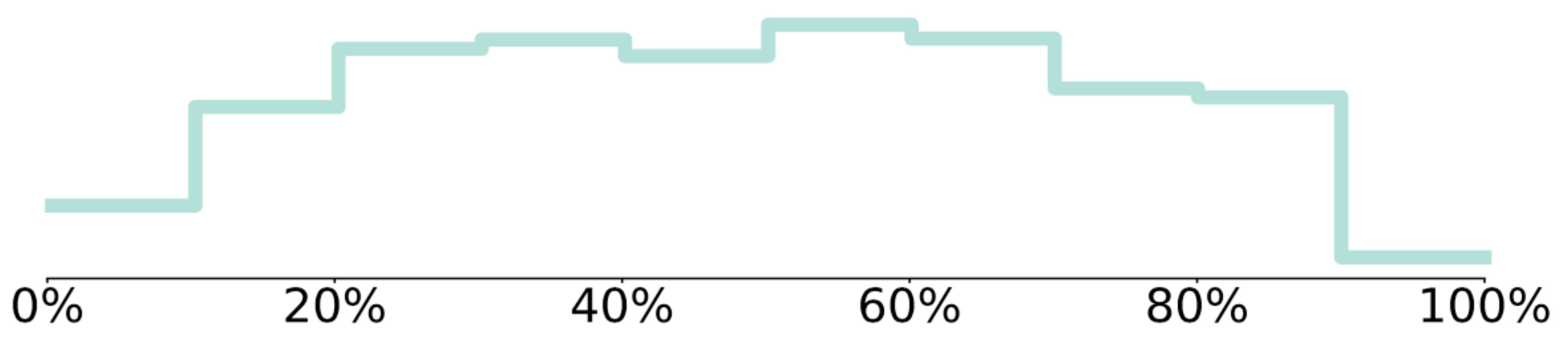
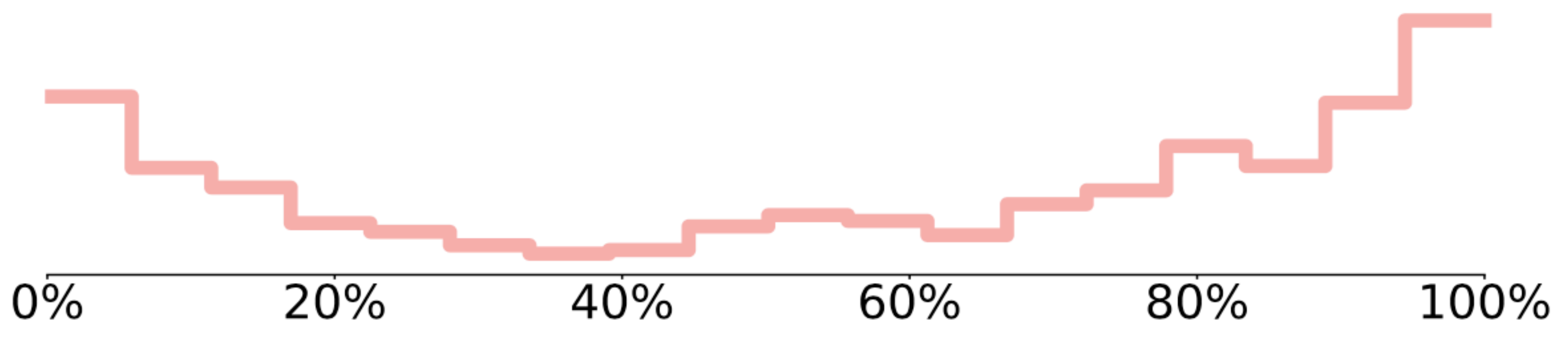
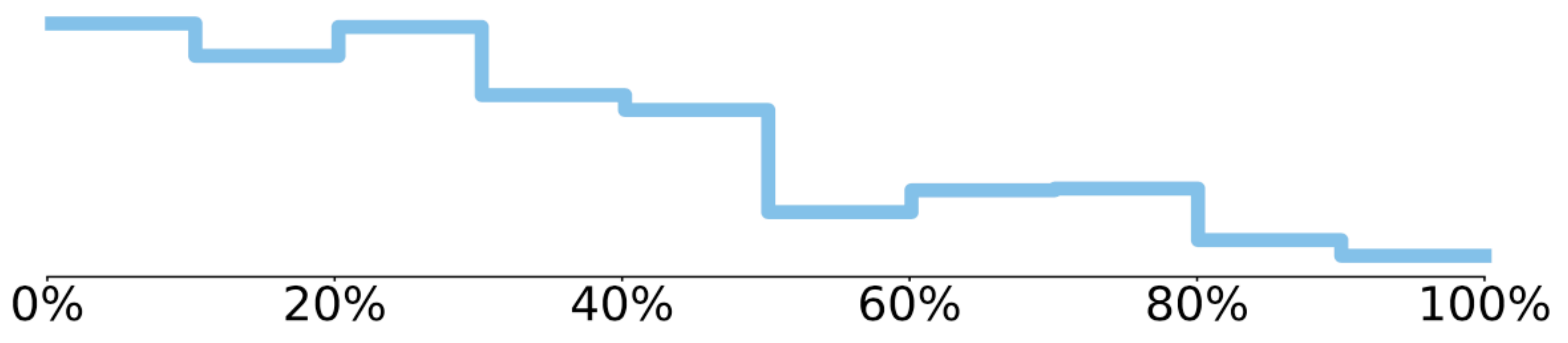
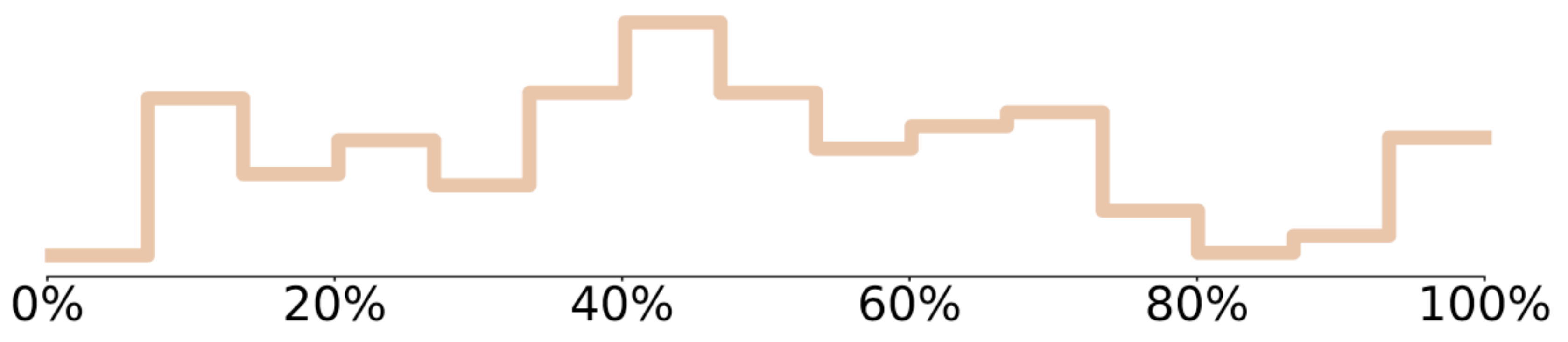
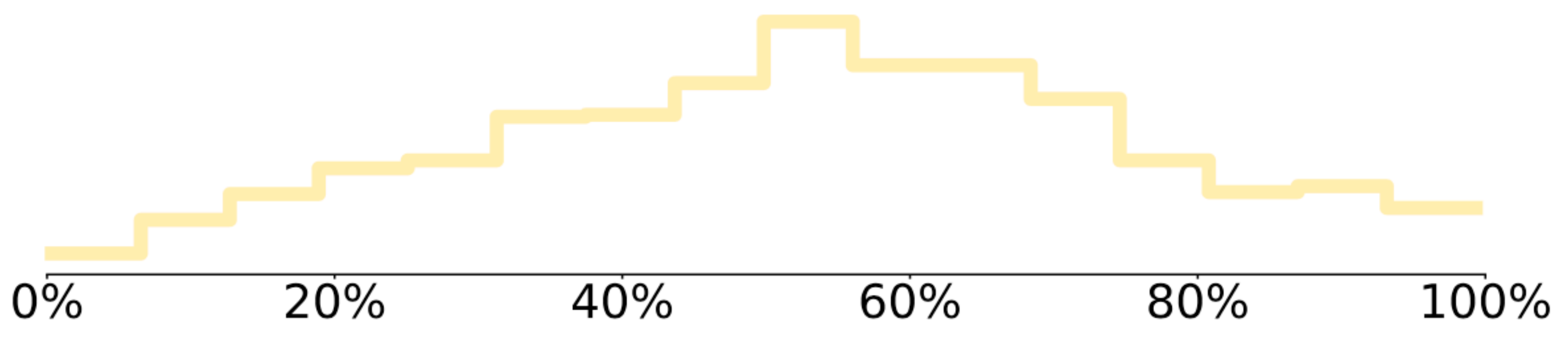
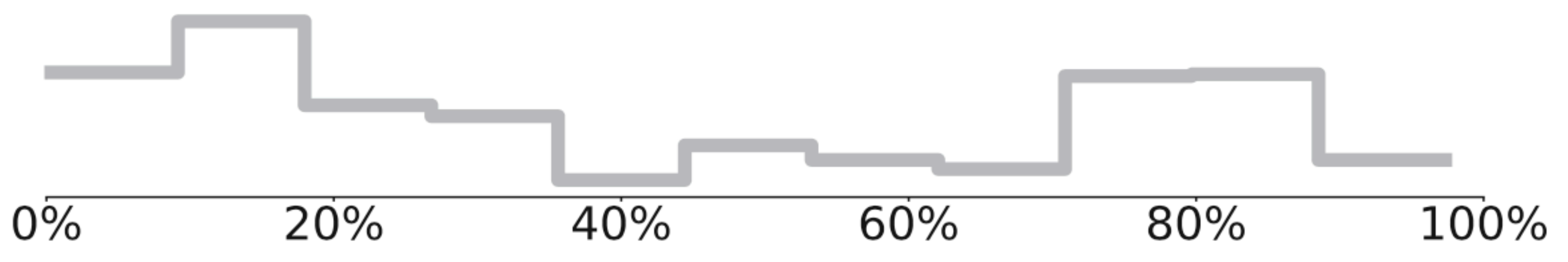


Table 7 The distribution of debugging activities per episode. % of episode time is the fraction of time of the episodes that the activity occupied

Activities	Instances Per Episode			% of Episode Time		
	Median (IQR)	Min	Max	Median (IQR)	Min	Max
Edit	3 (1-9)	0	67	41% (26-54%)	7%	97%
Test	3 (2-7)	0	32	29% (18-43%)	2%	100%
Navigate	3 (0-6)	0	109	15% (9-22%)	1%	50%
Inspect	0 (0-1)	0	26	14% (8-29%)	1%	58%
Consult	0 (0-1)	0	16	9% (4-18%)	0.4%	59%
Miscellaneous	0 (0-1)	0	35	4% (2-9%)	1%	26%

Alaboudi, A., LaToza, T.D. What constitutes debugging? An exploratory study of debugging episodes. *Empir Software Eng* **28**, 117 (2023). https://doi.org/10.1007/s10664-023-10352-5

Table 8 The distribution of frequency (count) of activities throughout the debugging episodes

Activities	Distribution of Frequency Across Episodes Time
Edit	
Test	
Navigate	
Inspect	
Consult	
Miscellaneous	

Alaboudi, A., LaToza, T.D. What constitutes debugging? An exploratory study of debugging episodes. *Empir Software Eng* **28**, 117 (2023). https://doi.org/10.1007/s10664-023-10352-5

Information needs in debugging

✱ *How did this **runtime state** occur? (12)*

data, memory corruption, race conditions, hangs, crashes, failed API calls, test failures, null pointers

✱ *Where was this **variable** last changed? (1)*

✱ *Why **didn't** this happen? (3)*

✖ *How is this object **different** from that object? (1)*

✖ *Which **team's** component caused this bug? (1)*

Which team should I assign this bug to?

✖ *What runtime state **changed** when this executed? (2)*

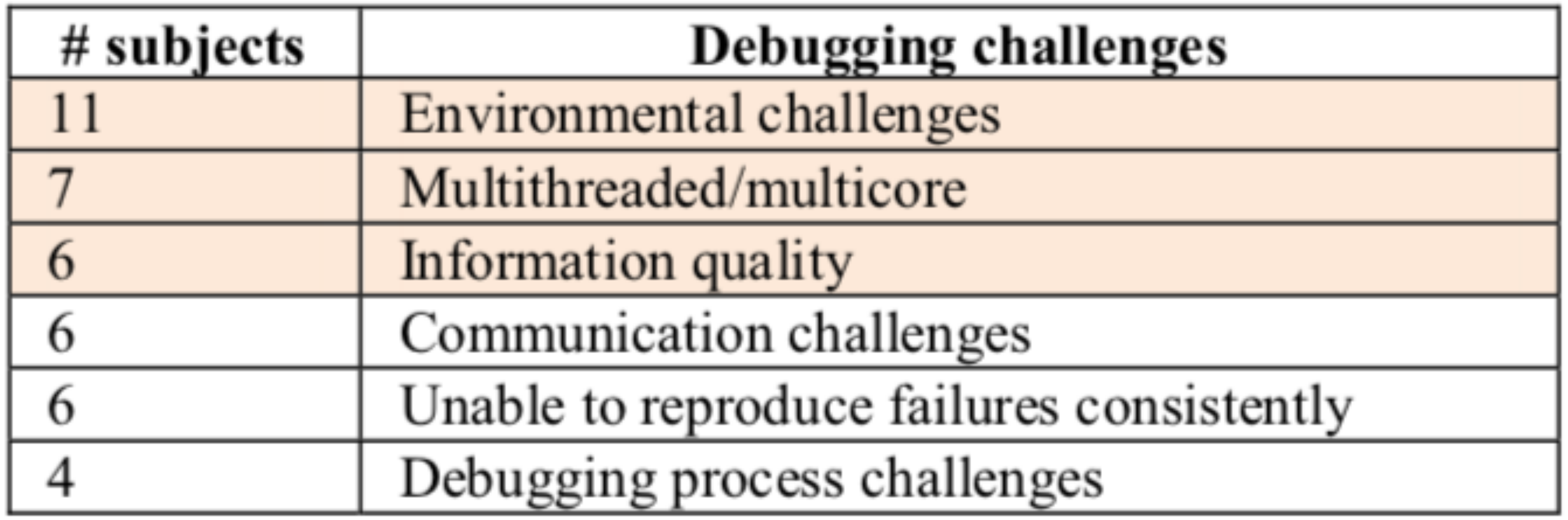
LaToza and Myers. Hard-to-answer questions about code. PLATEAU 2010.

Activity

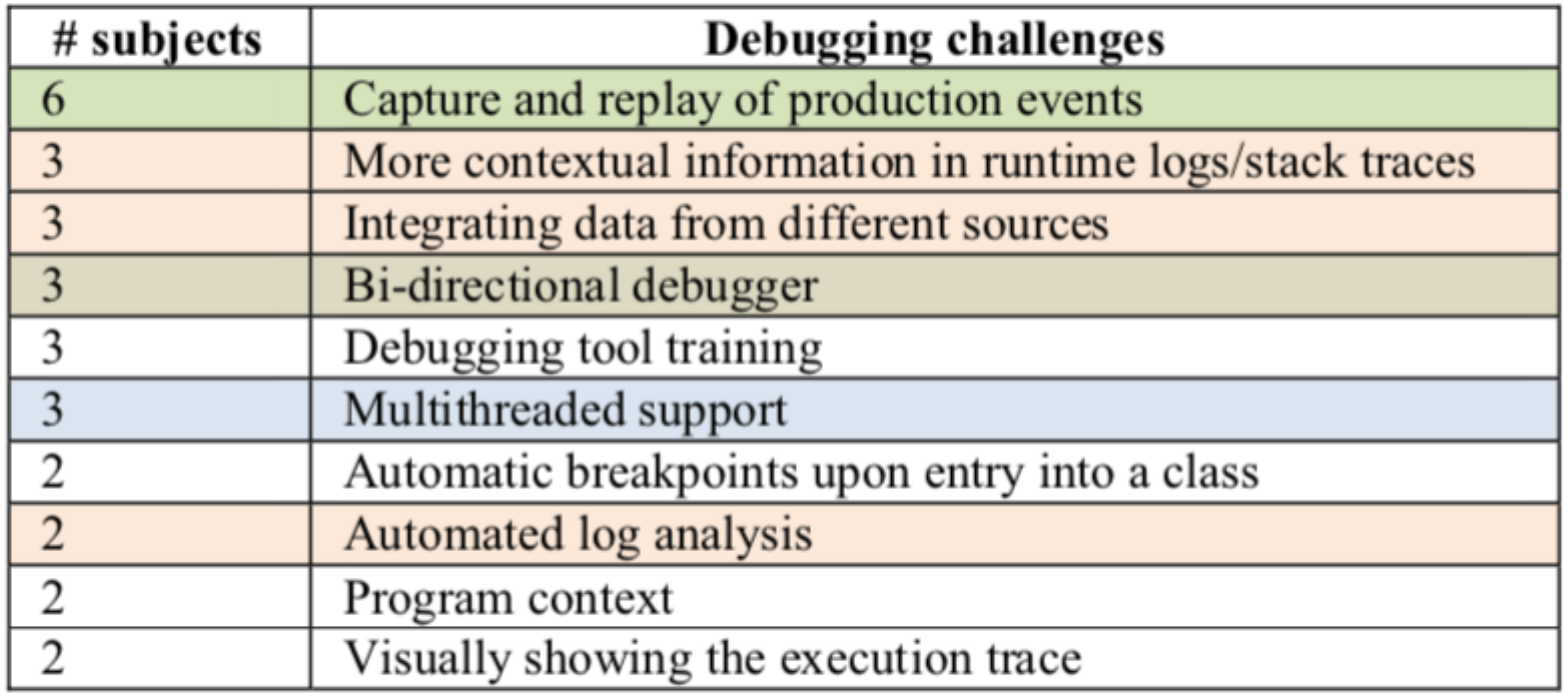
- What's the hardest debugging bug you've ever debugged?
- What made it hard?

What makes debugging hard?

- It may be unclear where behavior is implemented in code
- Fault may occur far away from failure
  - How to find connection?
- Understanding why failure occurred may be challenging
- Concurrency



# subjects	Debugging challenges
11	Environmental challenges
7	Multithreaded/multicore
6	Information quality
6	Communication challenges
6	Unable to reproduce failures consistently
4	Debugging process challenges



# subjects	Debugging challenges
6	Capture and replay of production events
3	More contextual information in runtime logs/stack traces
3	Integrating data from different sources
3	Bi-directional debugger
3	Debugging tool training
3	Multithreaded support
2	Automatic breakpoints upon entry into a class
2	Automated log analysis
2	Program context
2	Visually showing the execution trace

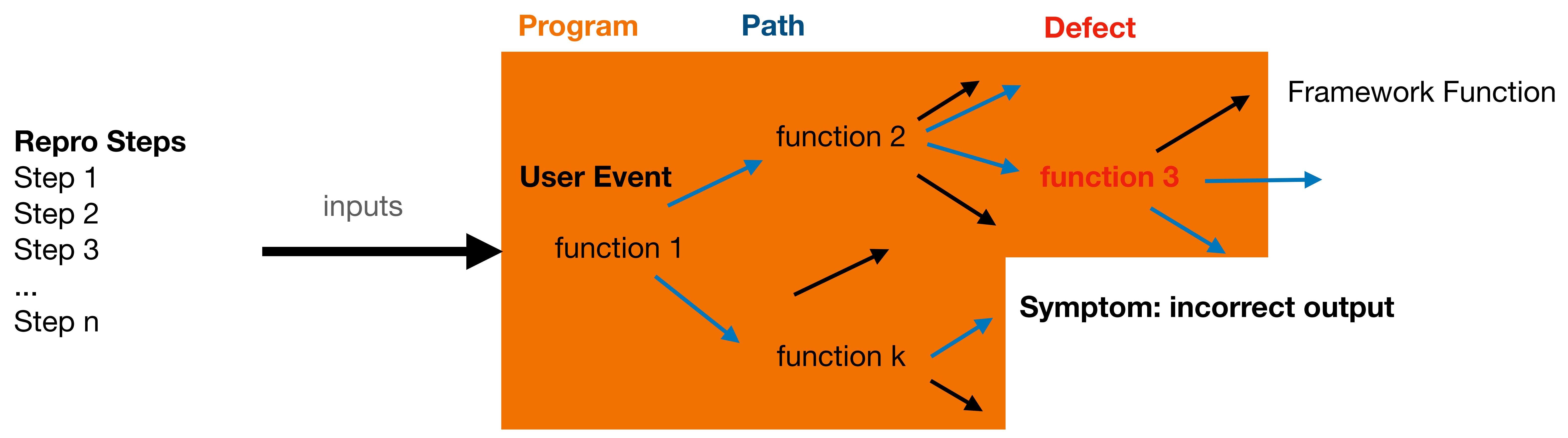
L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline and G. Venolia, "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers," *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, MD, 2013, pp. 383-392.

What makes hard bugs hard to debug?

- Cause / effect chasm - symptom far removed from the root cause (15 instances)
  - timing / synchronization problems
  - intermittent / inconsistent / infrequent bugs
  - materialize many iterations after root cause
  - uncertain connection to hardware / compiler / configuration
- Inapplicable tools (12 instances)
  - Heisenbugs - bug disappears when using debugging tool
  - long run to replicate - debugging tool slows down long run even more
  - stealth bug - bug consumes evidence to detect bug
  - context - configuration / memory makes it impossible to use tool
- What you see if probably illusory (7 instances)
  - misreads something in code or in runtime observations
- Faulty assumption (6)
- Spaghetti code (3)

Eisenstadt, M. Tales of Debugging from the Front Lines. Proc. Empirical Studies of Programmers, Ablex Publishing, Norwood, NJ, 1993, 86-112.

Model of Debugging Scenario

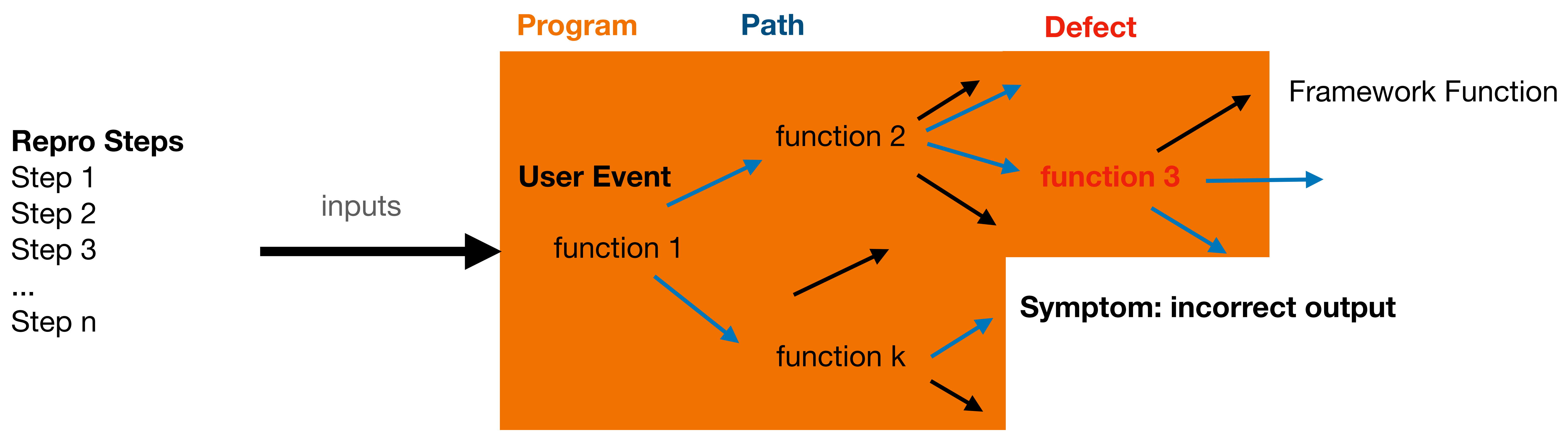


Traditional debugging tools

- Stepping in debugger
- Logging - insert print statements or wrap particular suspect functions
- Dump & diff - use diff tool to compare logging data between executions
- Conditional breakpoints
- Profiling tool - detect memory leaks, illegal memory references

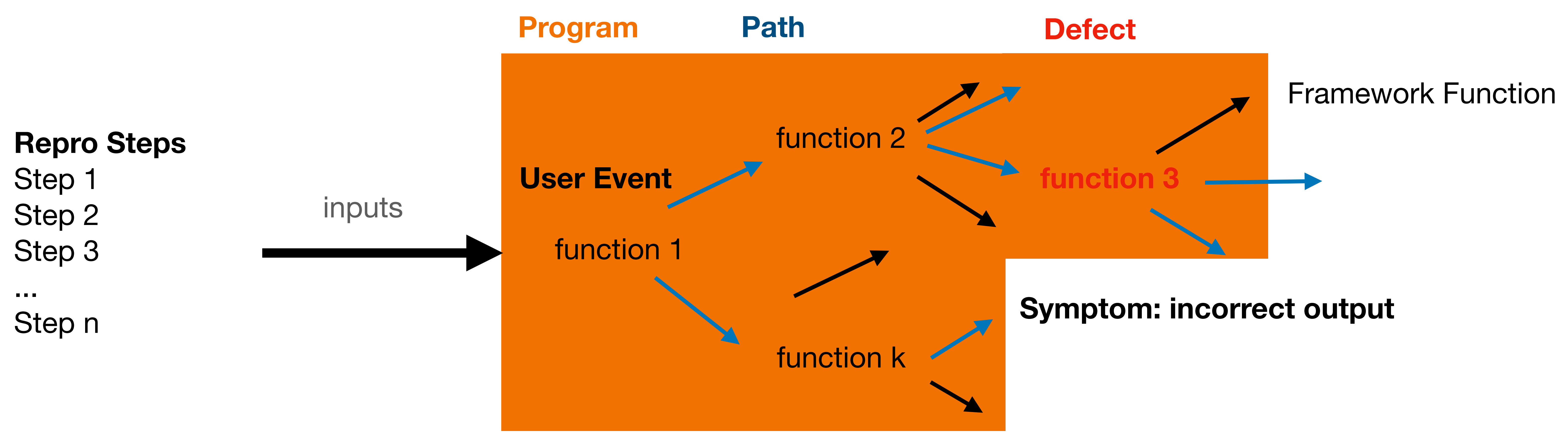
Eisenstadt, M. Tales of Debugging from the Front Lines. Proc. Empirical Studies of Programmers, Ablex Publishing, Norwood, NJ, 1993, 86-112.

Debugging Strategies



- Forwards & backwards debugging
- Minimizing repro steps
- Black box debugging
- Hypothesis testing

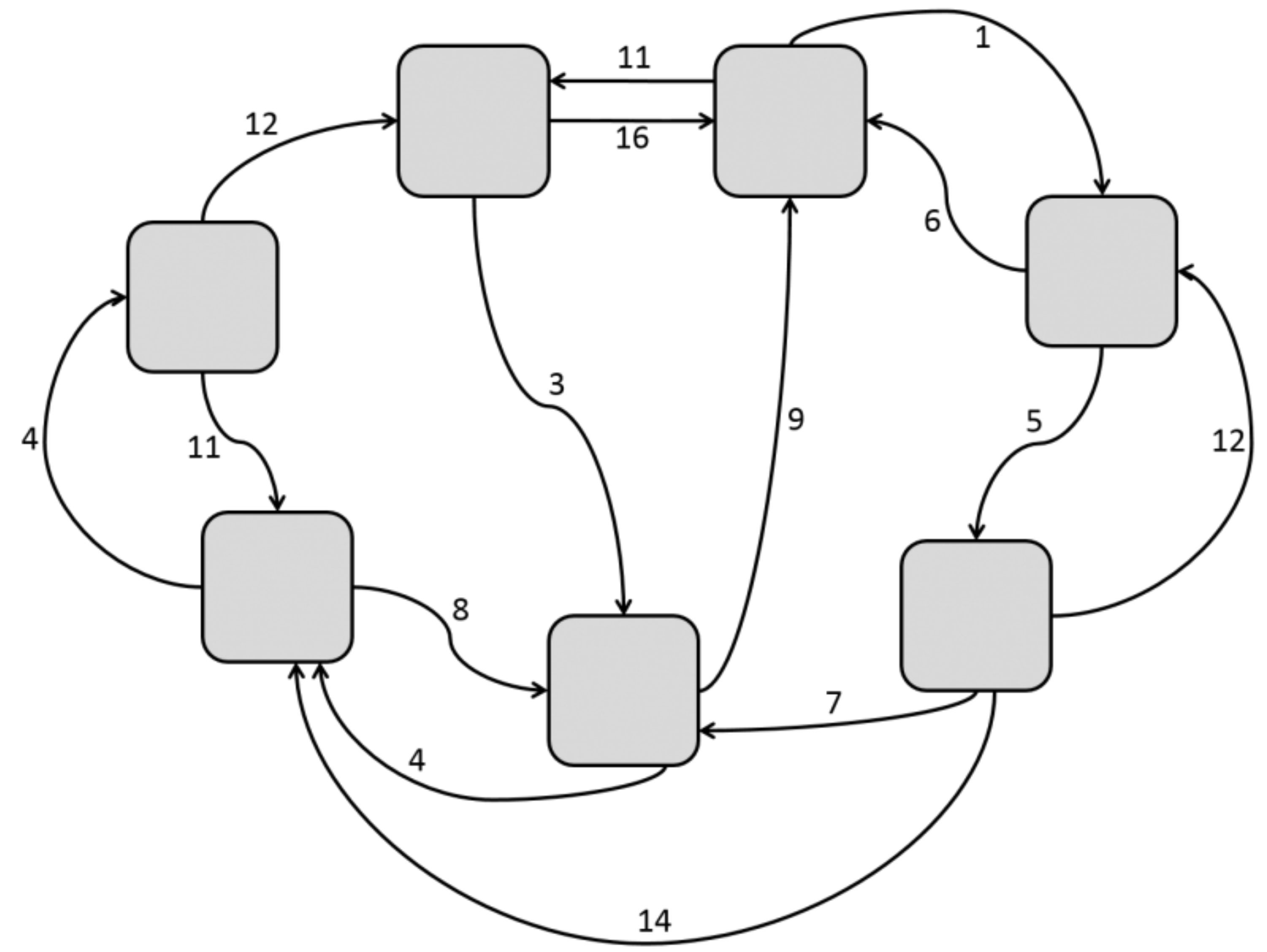
Information foraging



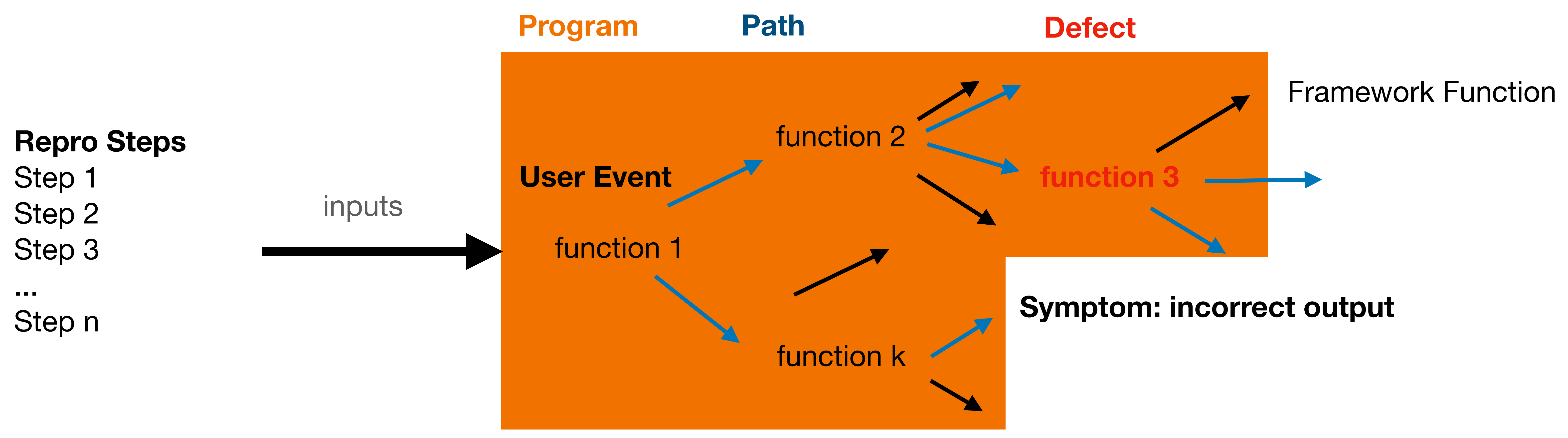
- Developers navigate call graph to find the defect.
- How? Information foraging
- Mathematical model describing navigation
- Analogy: animals foraging for food
  - Can forage in different patches (locations)
  - Goal is to maximize chances of finding **prey** while minimizing time spent in hunt (time debugging)
- Information foraging: navigating through an information space (patches) in order to maximize chances of finding prey (information) in minimal time

Information environment

- Information environment represented as **topology**
  - Information patches connected by traversable **links**
  - For SE, usually modeled as call graphs
    - methods are nodes and function invocations are edges

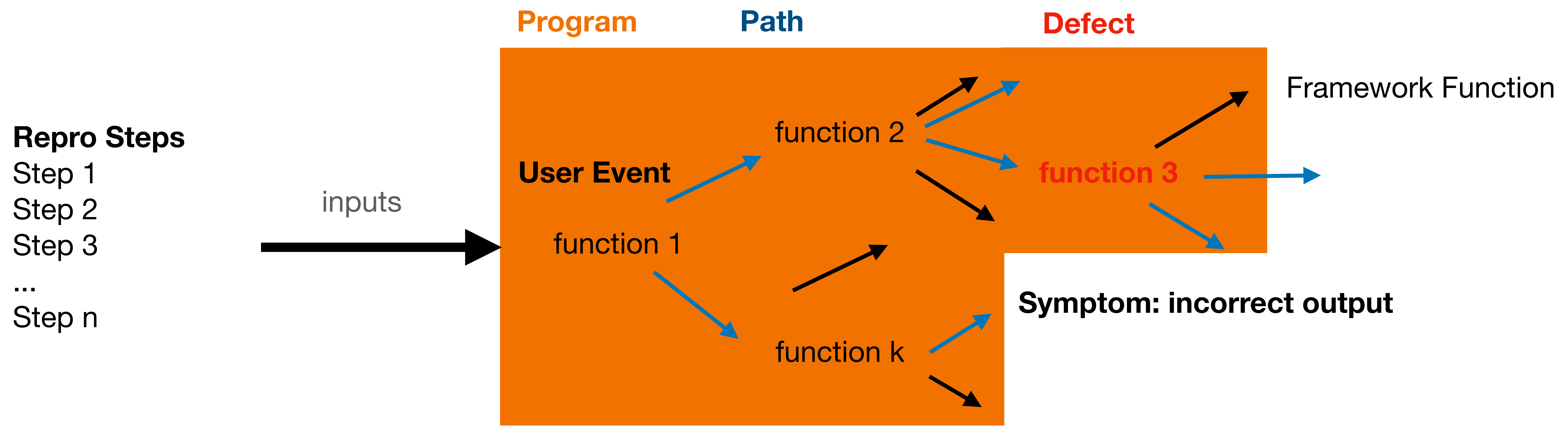


Traversing links



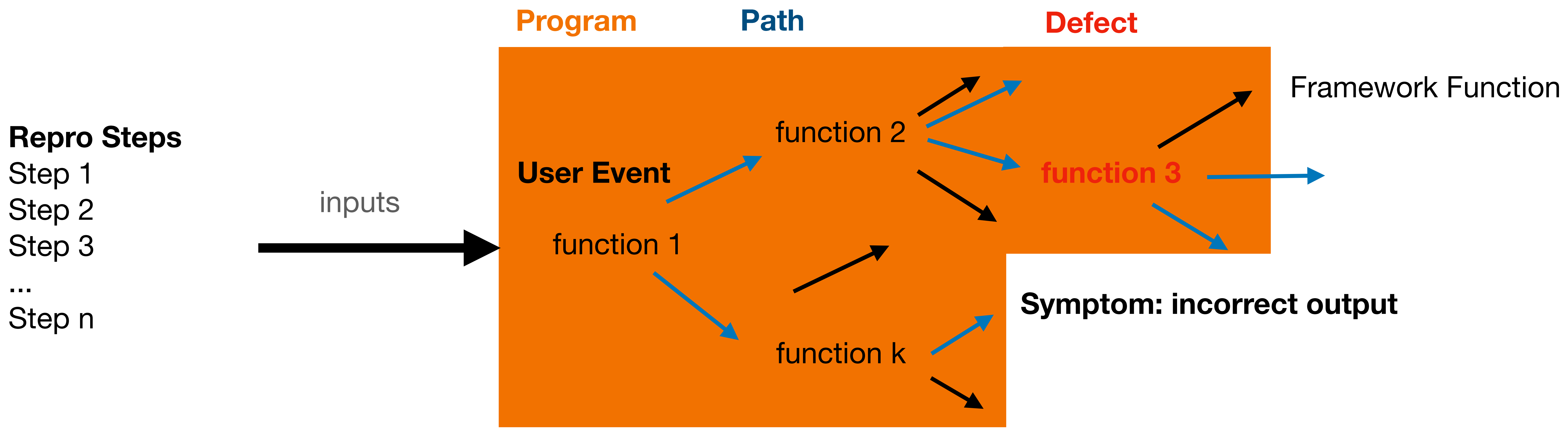
- Links - connection between patch offered by the information environment **(function call)**
- Cues - information features associated with outgoing links from patch **(function identifier)**
- User must choose which, of all possible links to traverse, has best chance of reaching prey

Scent



- User interprets cues on links by likelihood they will reach prey
    - e.g., do I think that the “invoke” method is likely to implement the functionality I’m looking for?

Debugging backwards



- What strategies might be used to more effectively navigate?
- Backwards slicing: start at output, systematically trace path backwards by asking what caused this
  - Why is x.color blue? Because it was set to blue by this assignment statement...
  - Can be hard
    - where is the statement that generated the output?
      - easier with an error, stack trace, breakpoint....
      - what happens if the output *didn't* get generated?
    - can't go backwards step by step in debugger
      - may have to guess at path, add print statements
      - or rerun program in debugger....

Find part of the program that caused incorrect output

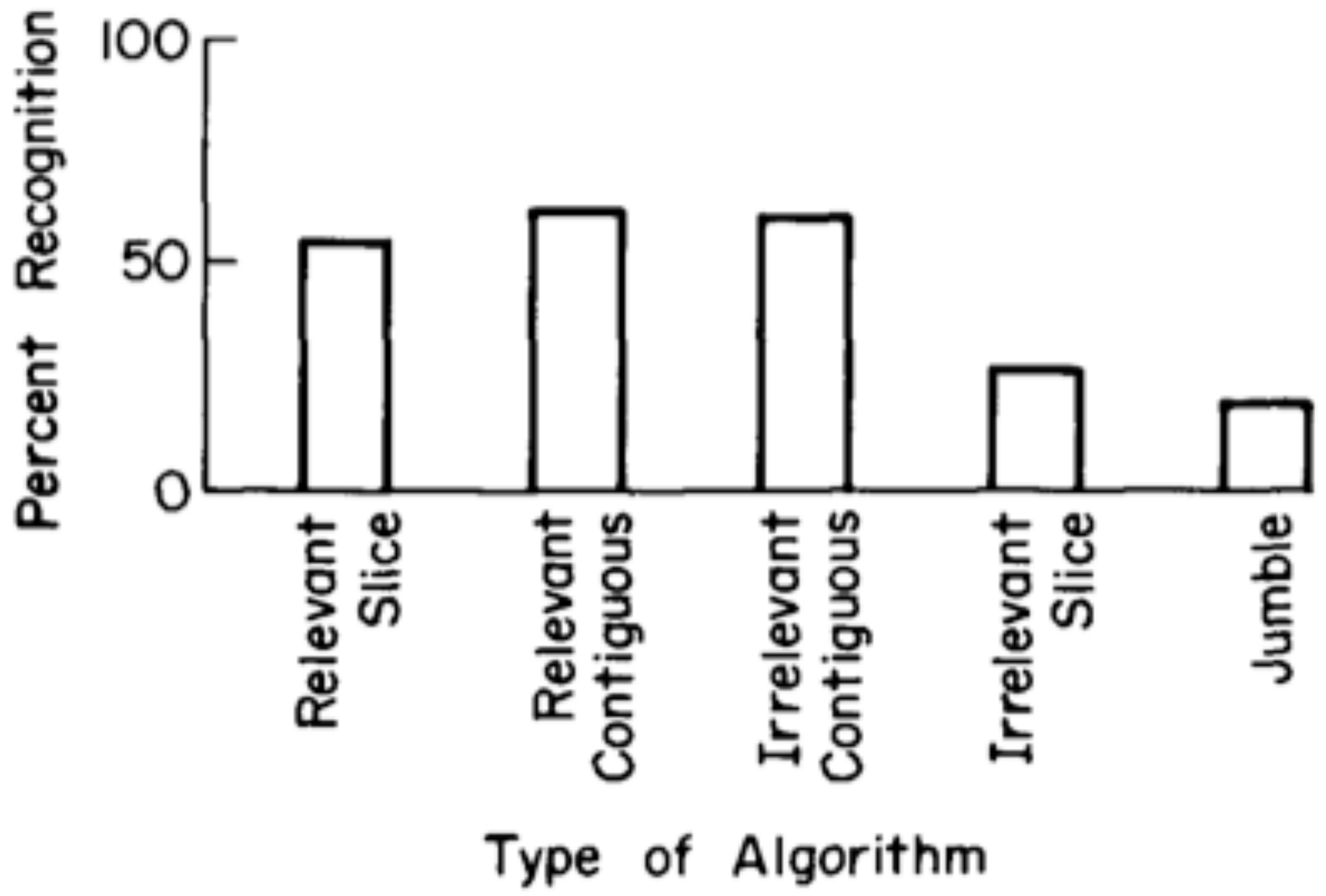
- Slice
  - Subset of the program that is responsible for computing the value of a variable at a program point
- Backwards slice
  - Transitive closure of all statements that have a control or data dependency
- Originally formulated as **subset** of program

Early evidence for slicing

Participants performed 3 debugging tasks on short code snippets

Asked to recognize code snippets afterwards

-
  ```
  BEGIN
  READ(X, Y)
  TOTAL := 0.0
  SUM := 0.0
  IF X <= 1
      THEN SUM := Y
      ELSE BEGIN
          READ(Z)
          TOTAL := X * Y
          END
  WRITE(TOTAL, SUM)
  END
  ```



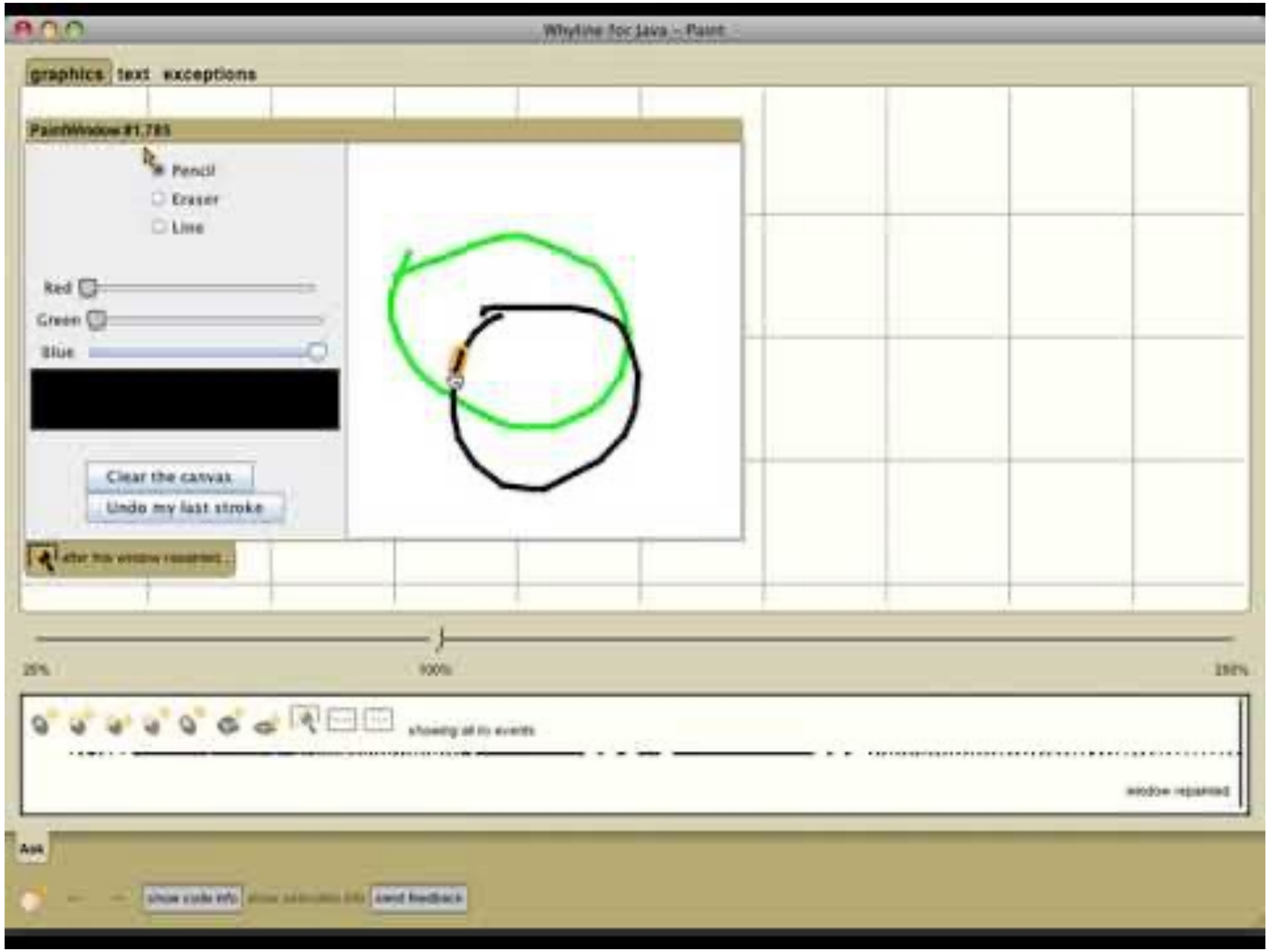
- (Static) slice - subset of the program that produces the same variable values at a program point
- Slice on variable Z at 12

Slicers debug faster

- Students debugging 100 LOC C++ programs
- Students given
  - Programming environment
  - Hardcopy input, wrong output, correct output
  - Files with program & input
- Compared students instructed to slice against everyone else
  - Excluding students who naturally use slicing strategy
- Slicers debug significantly faster (65.29 minutes vs. 30.16 minutes)

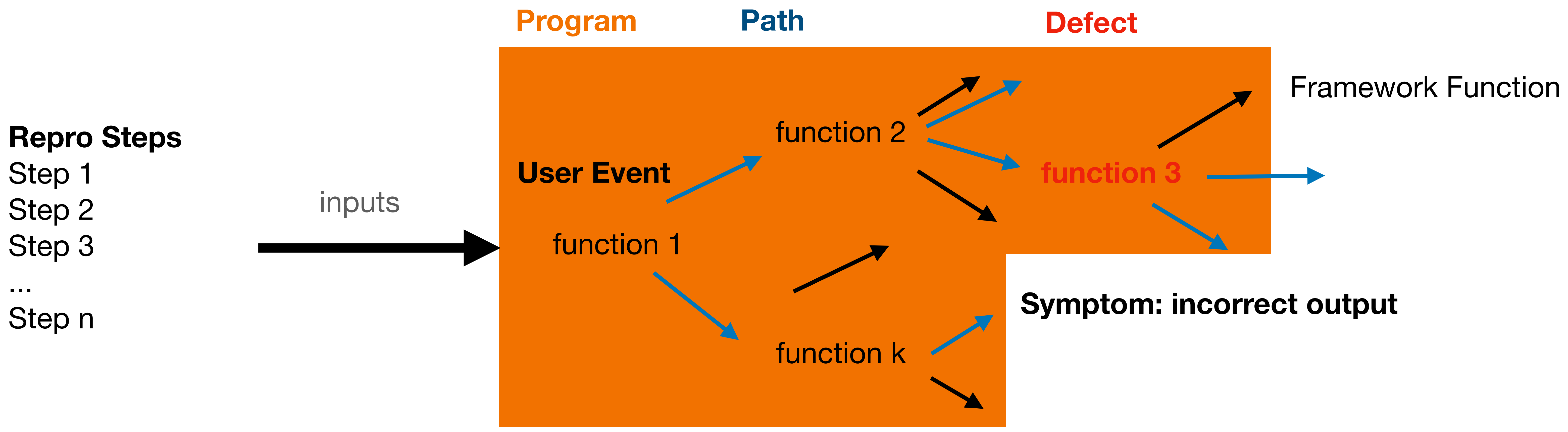
Francel M. A. and S. Rugaber (2001). The Value of Slicing While Debugging, *Science of Computer Programming*, 40(2-3), 151-169.

Associating incorrect output with responsible code



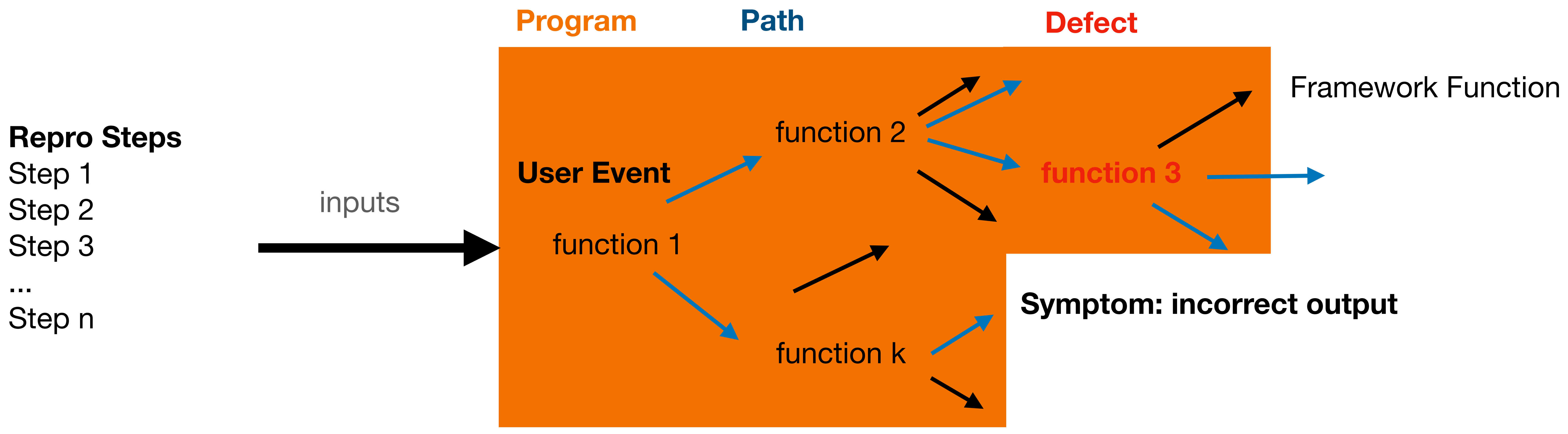
Amy J. Ko and Brad A. Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In Proceedings of the 30th international conference on Software engineering (ICSE '08). Association for Computing Machinery, New York, NY, USA, 301–310. https://doi.org/10.1145/1368088.1368130

Debugging forwards



- Forwards debugging: start at function before the defect, keep tracing forwards until find the defect
  - Can be hard
    - have to guess which statement will lead to defect
      - if guess wrong and miss the defect, have to start over

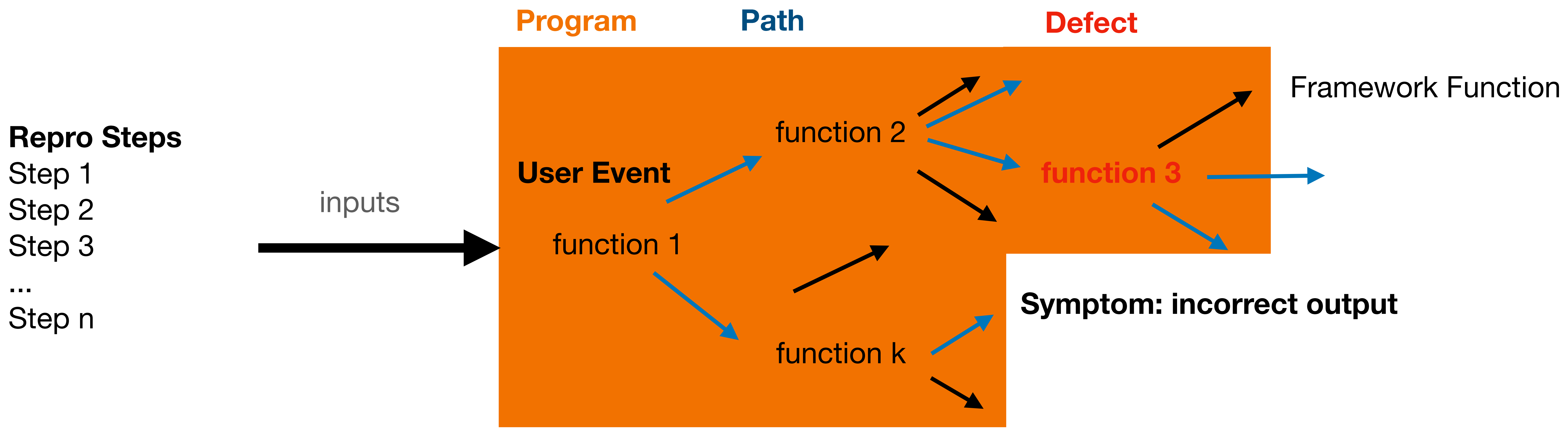
Minimizing repro steps: delta debugging



- Maybe the path is longer and more complex than it needs to be, making navigation hard.
- Is there a shorter and simpler path that still surfaces the defect?
- Need to find a simpler set of repro steps that still generates incorrect output
- Idea: delta debugging
  - Try each half of the repro steps, if neither fails, try cutting something smaller

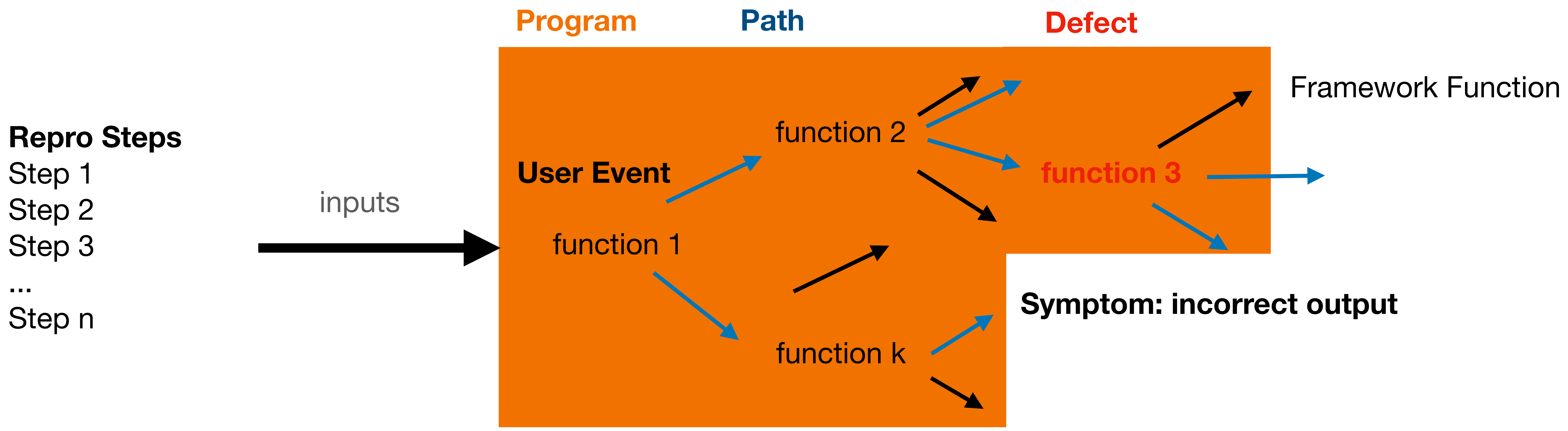
Zeller, Andreas (1999). "Yesterday, my program worked. Today, it does not. Why?". *Software Engineering — ESEC/FSE '99.* Lecture Notes in Computer Science. Vol. 1687. Springer. pp. 253–267. doi:10.1007/3-540-48166-4_16. ISBN 978-3-540-66538-0.

Black box debugging



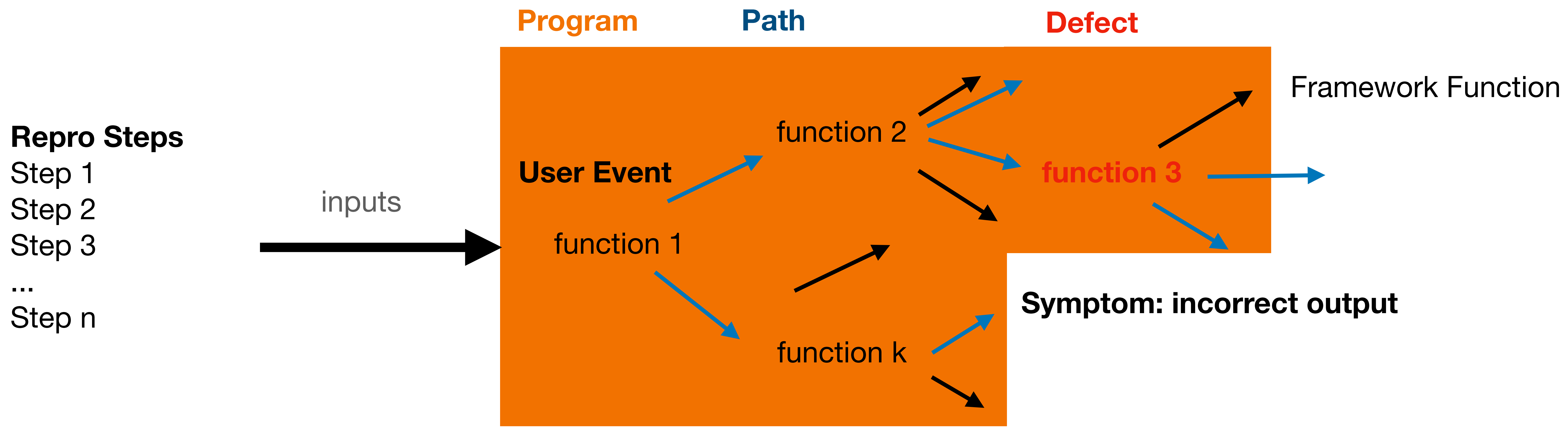
- What if the issue has already been localized, but it involves a framework function behaving unexpectedly
  - Don't really want to use backwards / forwards debugging to trace paths through a framework
    - Framework is a black box, can only be understood through its interface
  - Rather than trace through framework code, instead need to find strategies to reason about inputs/outputs with the framework

Black box debugging



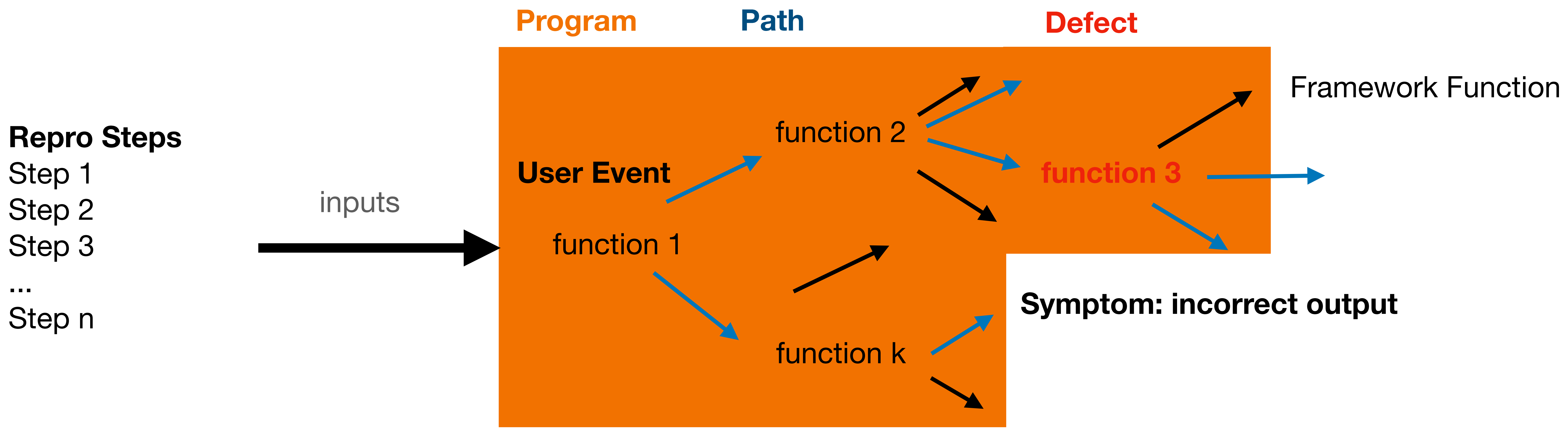
- StackOverflow
  - Maybe someone else used the framework incorrectly in the same way, and got a similar error message
  - Hard to use when
    - Not a unique error message
    - Unclear which (of many) framework interactions are relevant

Black box debugging



- Simplify framework interactions
  - Maybe the framework interactions are all wrong
  - Try with a simpler framework interaction that is correct (e.g., an example app) and see if that works

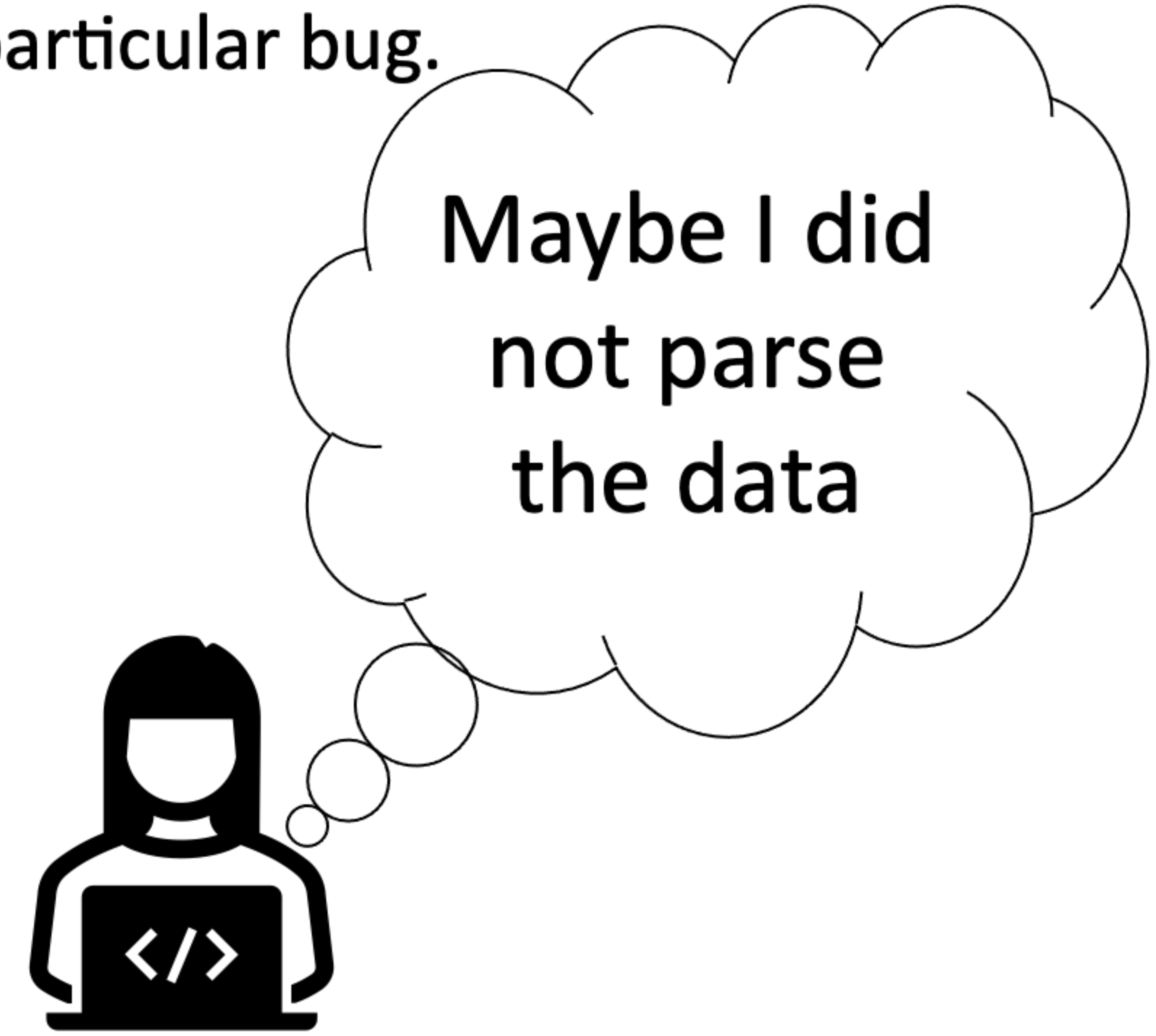
Hypothesis testing



- StackOverflow
  - Maybe someone else used the framework incorrectly in the same way, and got a similar error message
  - Hard to use when
    - Not a unique error message
    - Unclear which (of many) framework interactions are relevant

Hypothesis testing

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



Work to test the hypothesis

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

Formulate & test hypotheses

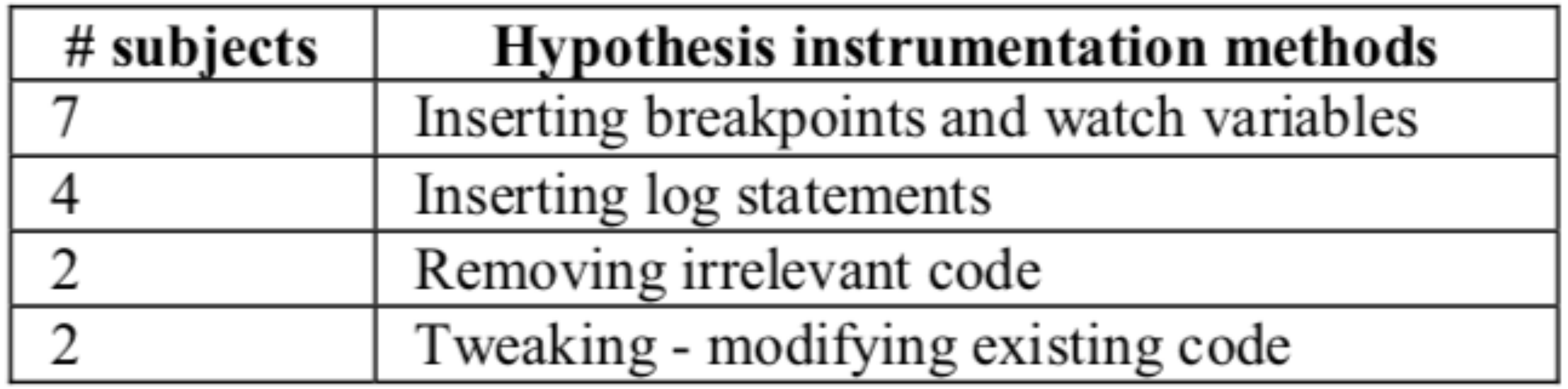
- Use knowledge & data so far to formulate hypothesis about why bug happened with
  - cogitation, meditation, observation, inspection, contemplation, hand-simulation, gestation, rumination, dedication, inspiration, articulation

- Recognize cliche
  seen a similar bug before

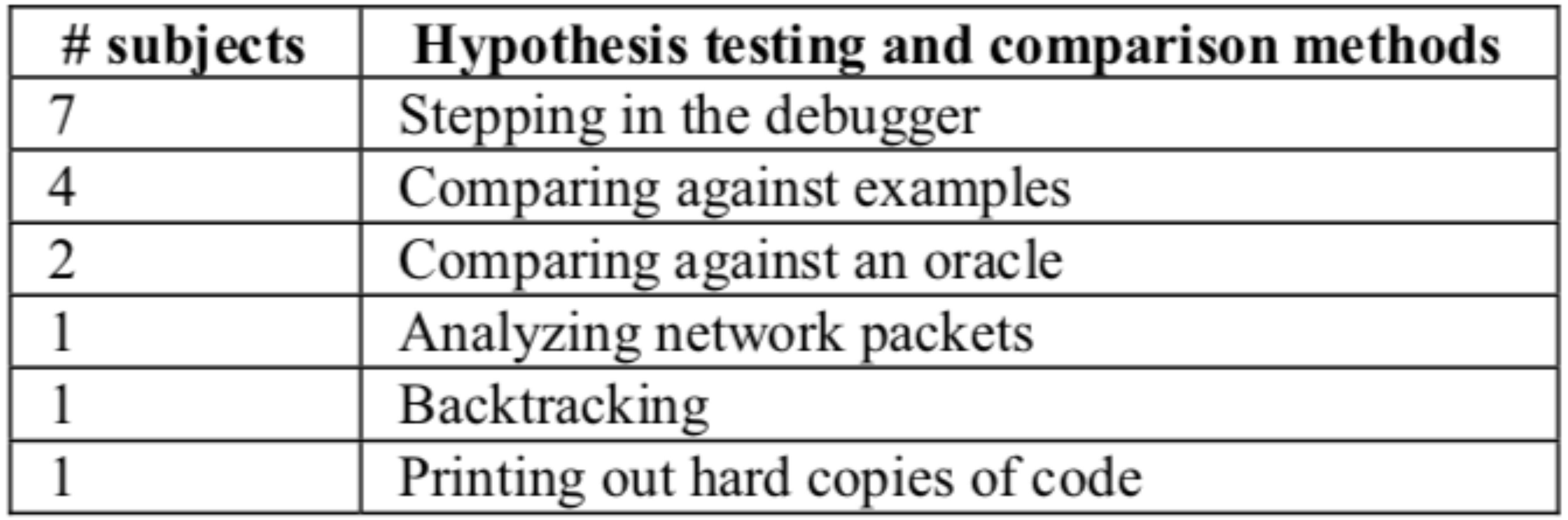
- Controlled experiments - test hypotheses by gathering data

Eisenstadt, M. Tales of Debugging from the Front Lines. Proc. Empirical Studies of Programmers, Ablex Publishing, Norwood, NJ, 1993, 86-112.

Resources for testing hypotheses



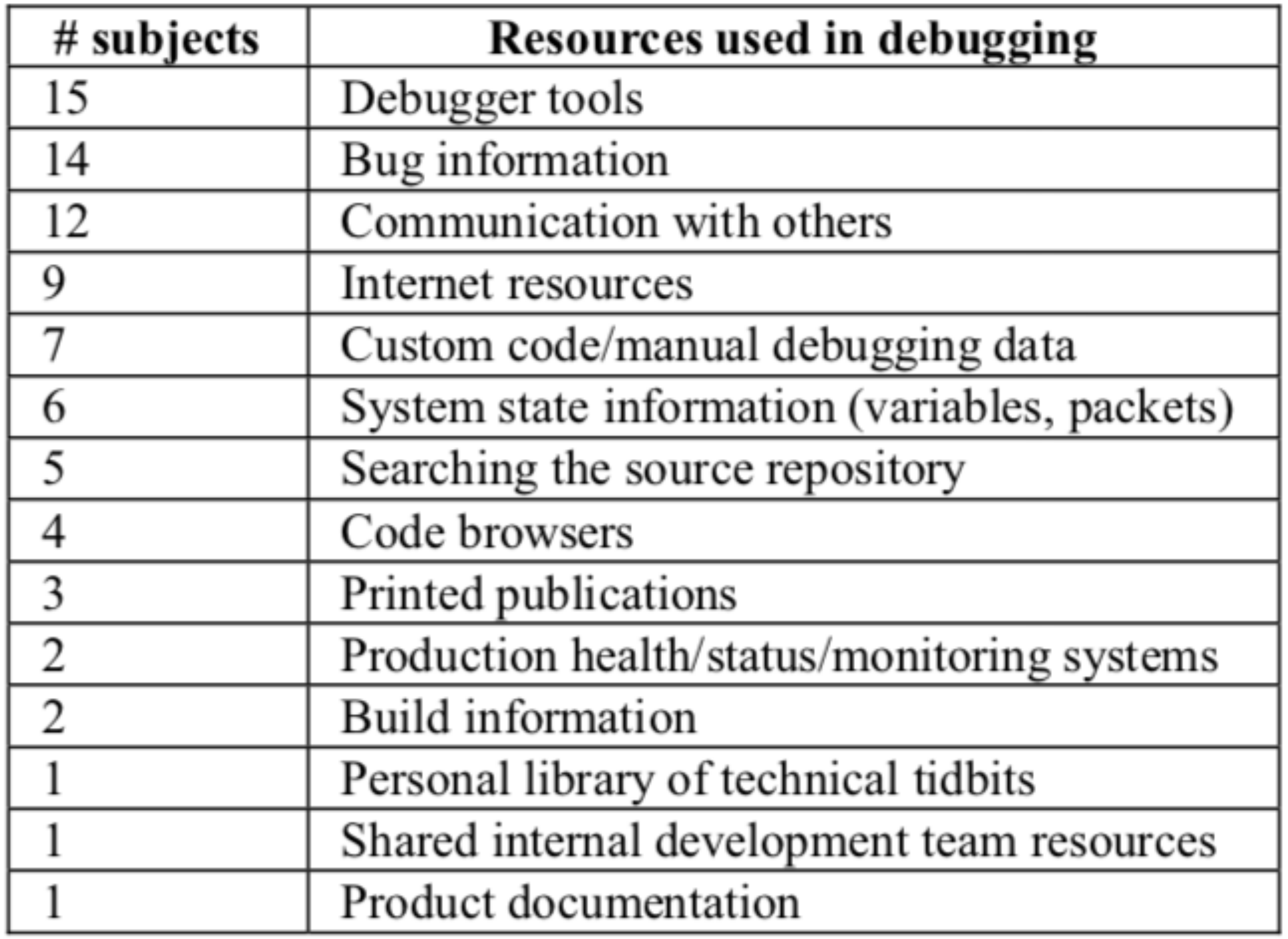
# subjects	Hypothesis instrumentation methods
7	Inserting breakpoints and watch variables
4	Inserting log statements
2	Removing irrelevant code
2	Tweaking - modifying existing code



# subjects	Hypothesis testing and comparison methods
7	Stepping in the debugger
4	Comparing against examples
2	Comparing against an oracle
1	Analyzing network packets
1	Backtracking
1	Printing out hard copies of code

L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline and G. Venolia, "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers," *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, MD, 2013, pp. 383-392.

Resources used in debugging

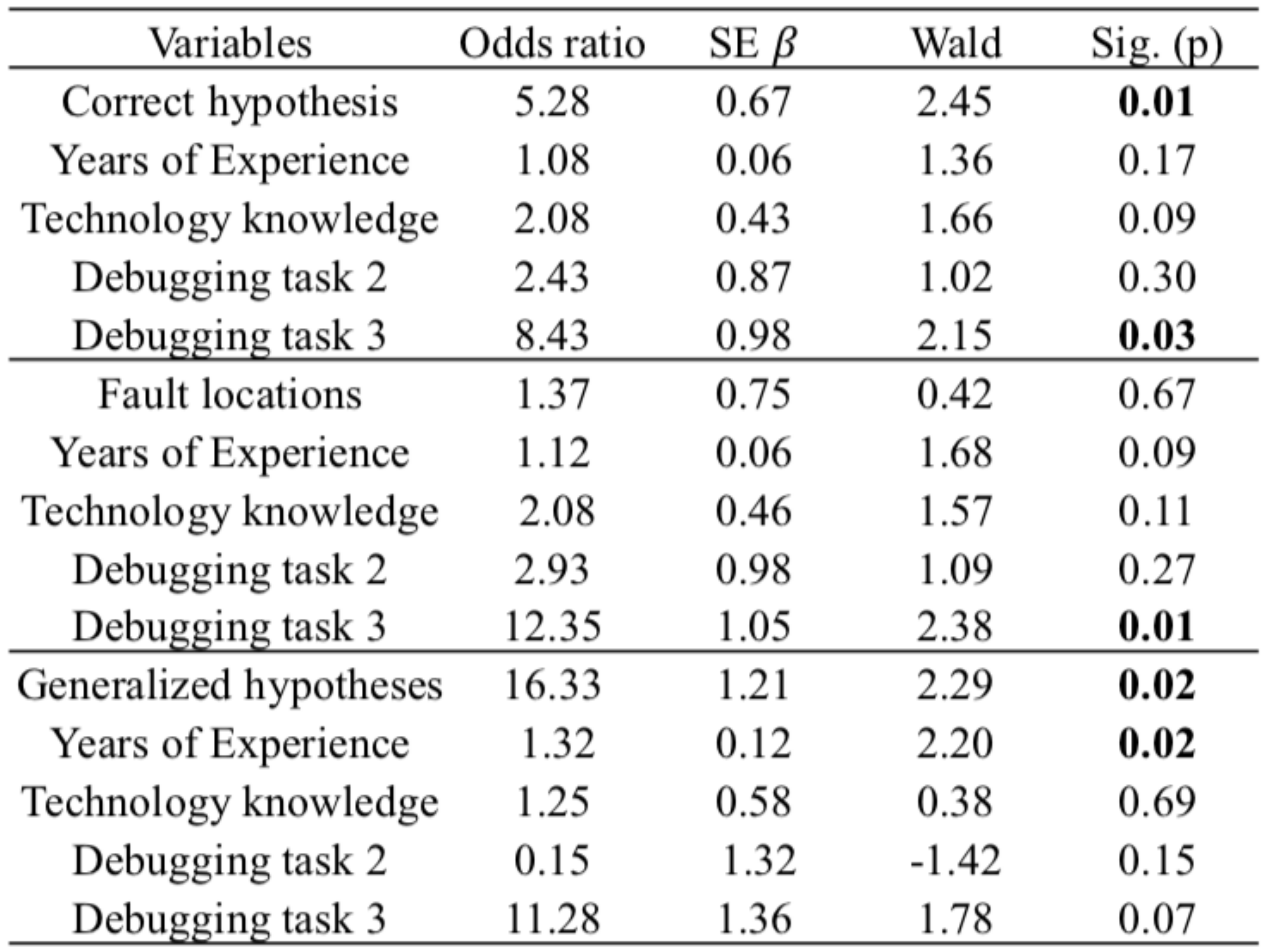


# subjects	Resources used in debugging
15	Debugger tools
14	Bug information
12	Communication with others
9	Internet resources
7	Custom code/manual debugging data
6	System state information (variables, packets)
5	Searching the source repository
4	Code browsers
3	Printed publications
2	Production health/status/monitoring systems
2	Build information
1	Personal library of technical tidbits
1	Shared internal development team resources
1	Product documentation

L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline and G. Venolia, "Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers," *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, MD, 2013, pp. 383-392.

Debugging hypotheses matter

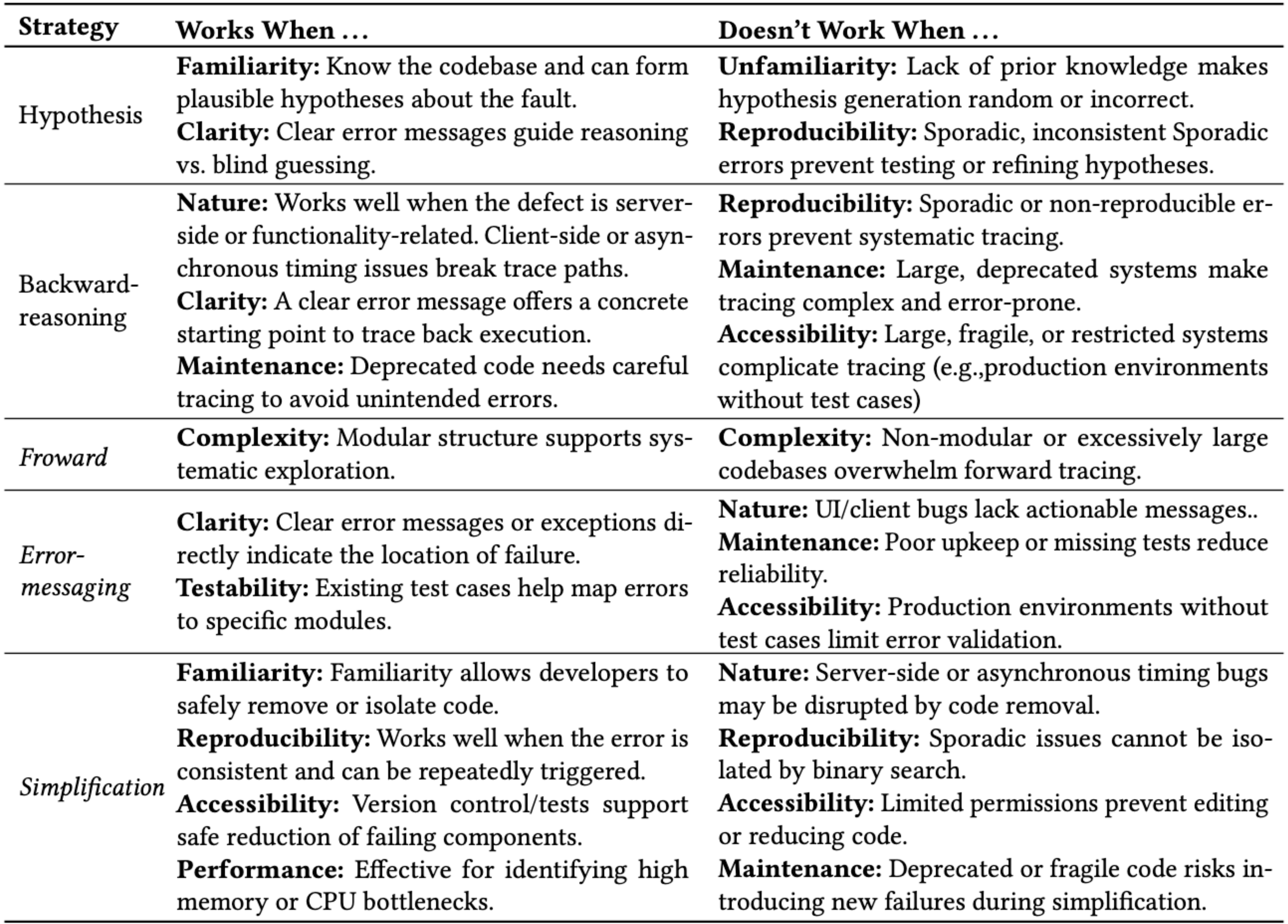
- Developers with a correct hypothesis early in the debugging process
  - Spent **30% less time** fixing the fault
  - **>5x** more likely to succeed
- No evidence industrial programming experience or more knowledge of related technologies associated with better hypotheses performance.
- **No evidence that providing potential fault locations helps debugging.**
- Providing generalized debugging hypotheses
  - **> 16x more likely** to successfully fix a fault



Variables	Odds ratio	SE β	Wald	Sig. (p)
Correct hypothesis	5.28	0.67	2.45	**0.01**
Years of Experience	1.08	0.06	1.36	0.17
Technology knowledge	2.08	0.43	1.66	0.09
Debugging task 2	2.43	0.87	1.02	0.30
Debugging task 3	8.43	0.98	2.15	**0.03**
Fault locations	1.37	0.75	0.42	0.67
Years of Experience	1.12	0.06	1.68	0.09
Technology knowledge	2.08	0.46	1.57	0.11
Debugging task 2	2.93	0.98	1.09	0.27
Debugging task 3	12.35	1.05	2.38	**0.01**
Generalized hypotheses	16.33	1.21	2.29	**0.02**
Years of Experience	1.32	0.12	2.20	**0.02**
Technology knowledge	1.25	0.58	0.38	0.69
Debugging task 2	0.15	1.32	-1.42	0.15
Debugging task 3	11.28	1.36	1.78	0.07

A. Alaboudi and T. D. LaToza, "Using Hypotheses as a Debugging Aid," *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Dunedin, New Zealand, 2020, pp. 1-9, doi: 10.1109/VL/HCC50065.2020.9127273.

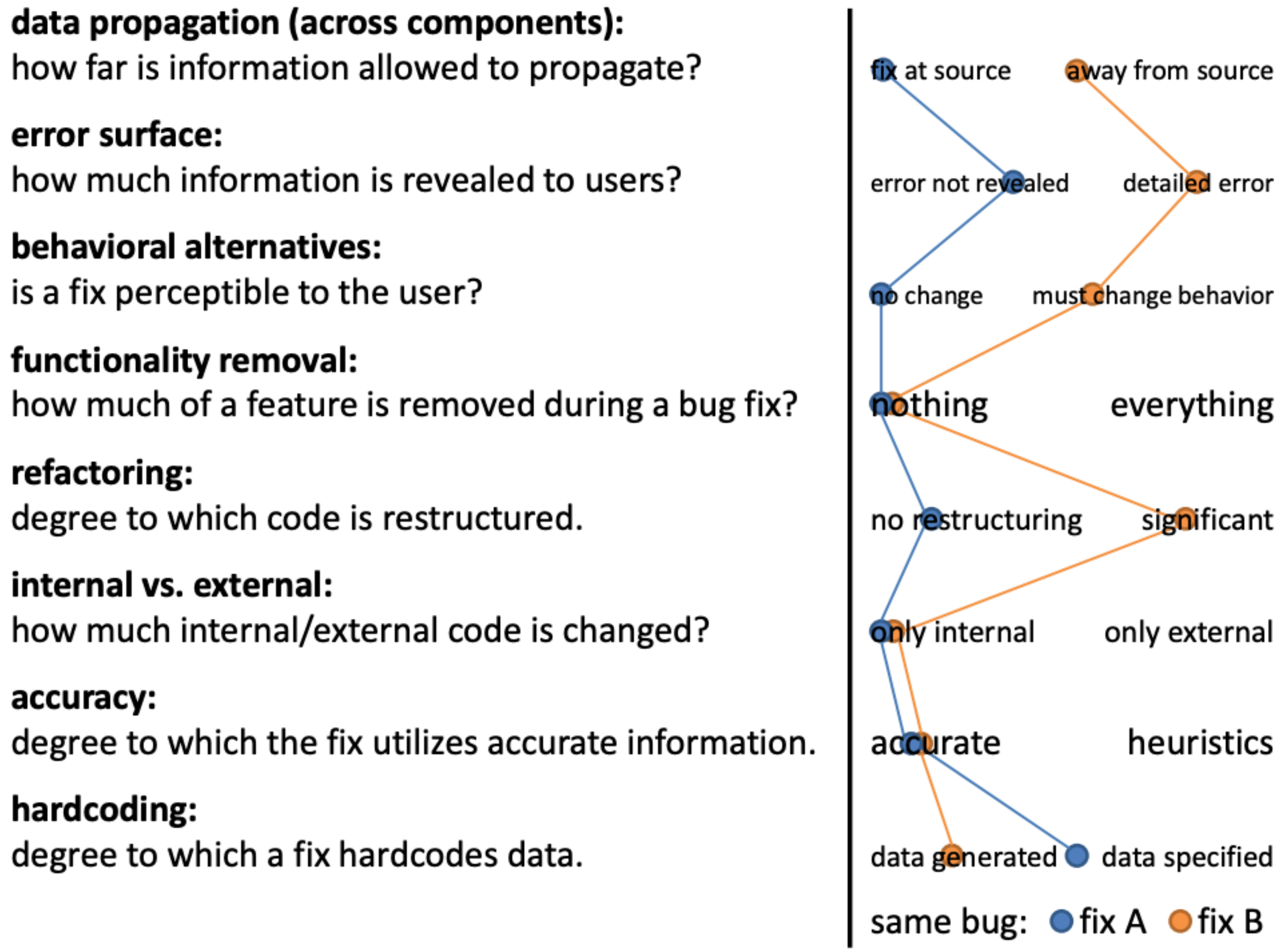
Factors impacting strategy choice



Strategy	Works When …	Doesn't Work When …
Hypothesis	**Familiarity:** Know the codebase and can form plausible hypotheses about the fault. **Clarity:** Clear error messages guide reasoning vs. blind guessing.	**Unfamiliarity:** Lack of prior knowledge makes hypothesis generation random or incorrect. **Reproducibility:** Sporadic, inconsistent Sporadic errors prevent testing or refining hypotheses.
Backward-reasoning	**Nature:** Works well when the defect is server-side or functionality-related. Client-side or asynchronous timing issues break trace paths. **Clarity:** A clear error message offers a concrete starting point to trace back execution. **Maintenance:** Deprecated code needs careful tracing to avoid unintended errors.	**Reproducibility:** Sporadic or non-reproducible errors prevent systematic tracing. **Maintenance:** Large, deprecated systems make tracing complex and error-prone. **Accessibility:** Large, fragile, or restricted systems complicate tracing (e.g.,production environments without test cases)
Froward	**Complexity:** Modular structure supports systematic exploration.	**Complexity:** Non-modular or excessively large codebases overwhelm forward tracing.
Error-messaging	**Clarity:** Clear error messages or exceptions directly indicate the location of failure. **Testability:** Existing test cases help map errors to specific modules.	**Nature:** UI/client bugs lack actionable messages.. **Maintenance:** Poor upkeep or missing tests reduce reliability. **Accessibility:** Production environments without test cases limit error validation.
Simplification	**Familiarity:** Familiarity allows developers to safely remove or isolate code. **Reproducibility:** Works well when the error is consistent and can be repeatedly triggered. **Accessibility:** Version control/tests support safe reduction of failing components. **Performance:** Effective for identifying high memory or CPU bottlenecks.	**Nature:** Server-side or asynchronous timing bugs may be disrupted by code removal. **Reproducibility:** Sporadic issues cannot be isolated by binary search. **Accessibility:** Limited permissions prevent editing or reducing code. **Maintenance:** Deprecated or fragile code risks introducing new failures during simplification.

Fixing defects

- Fault localization is only part of the debugging process
- After finding the defect, need to design a fix that addresses it
- Many design choices about how to do this effectively
- LLMs traditionally use test passing as way to evaluate fix, may not effectively address design considerations



data propagation (across components):
how far is information allowed to propagate?

error surface:
how much information is revealed to users?

behavioral alternatives:
is a fix perceptible to the user?

functionality removal:
how much of a feature is removed during a bug fix?

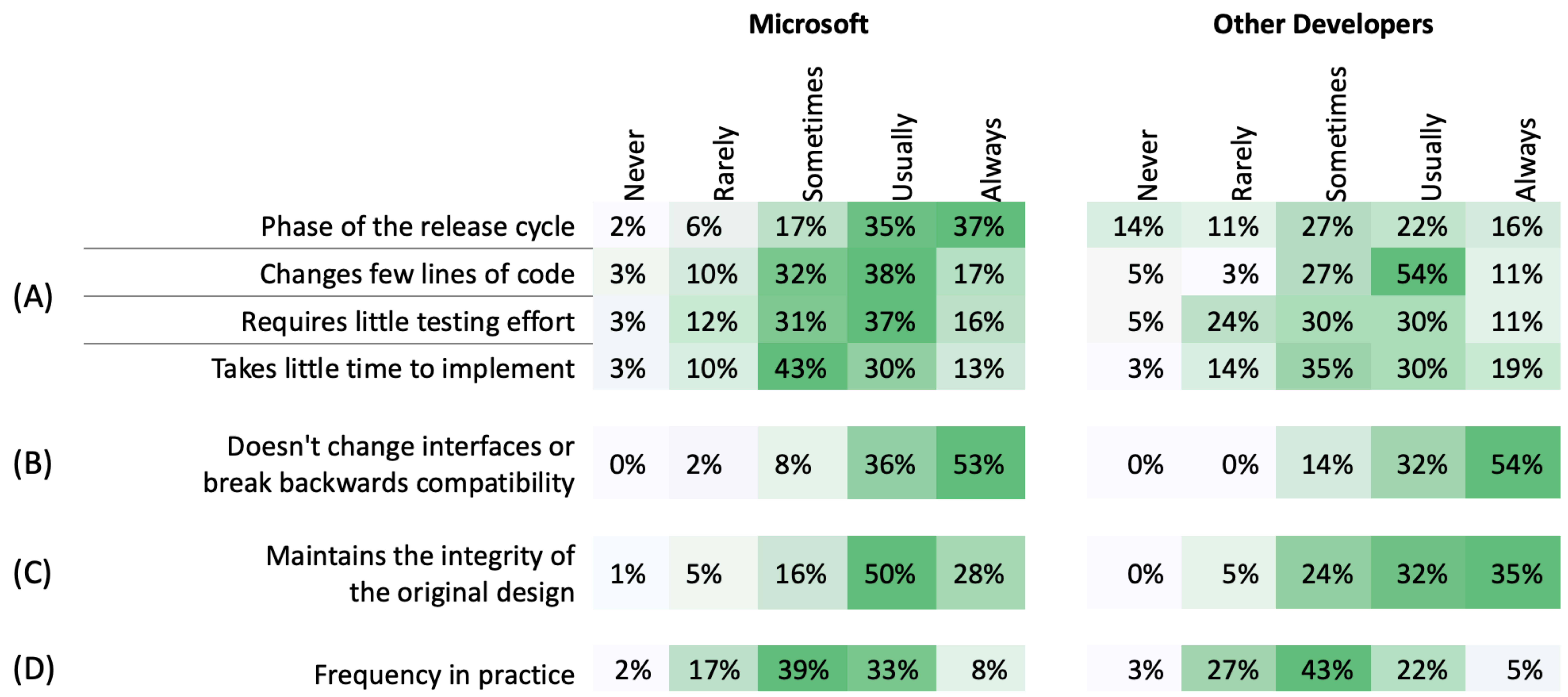
refactoring:
degree to which code is restructured.

internal vs. external:
how much internal/external code is changed?

accuracy:
degree to which the fix utilizes accurate information.

hardcoding:
degree to which a fix hardcodes data.

Factors that influence engineers' bug fix design



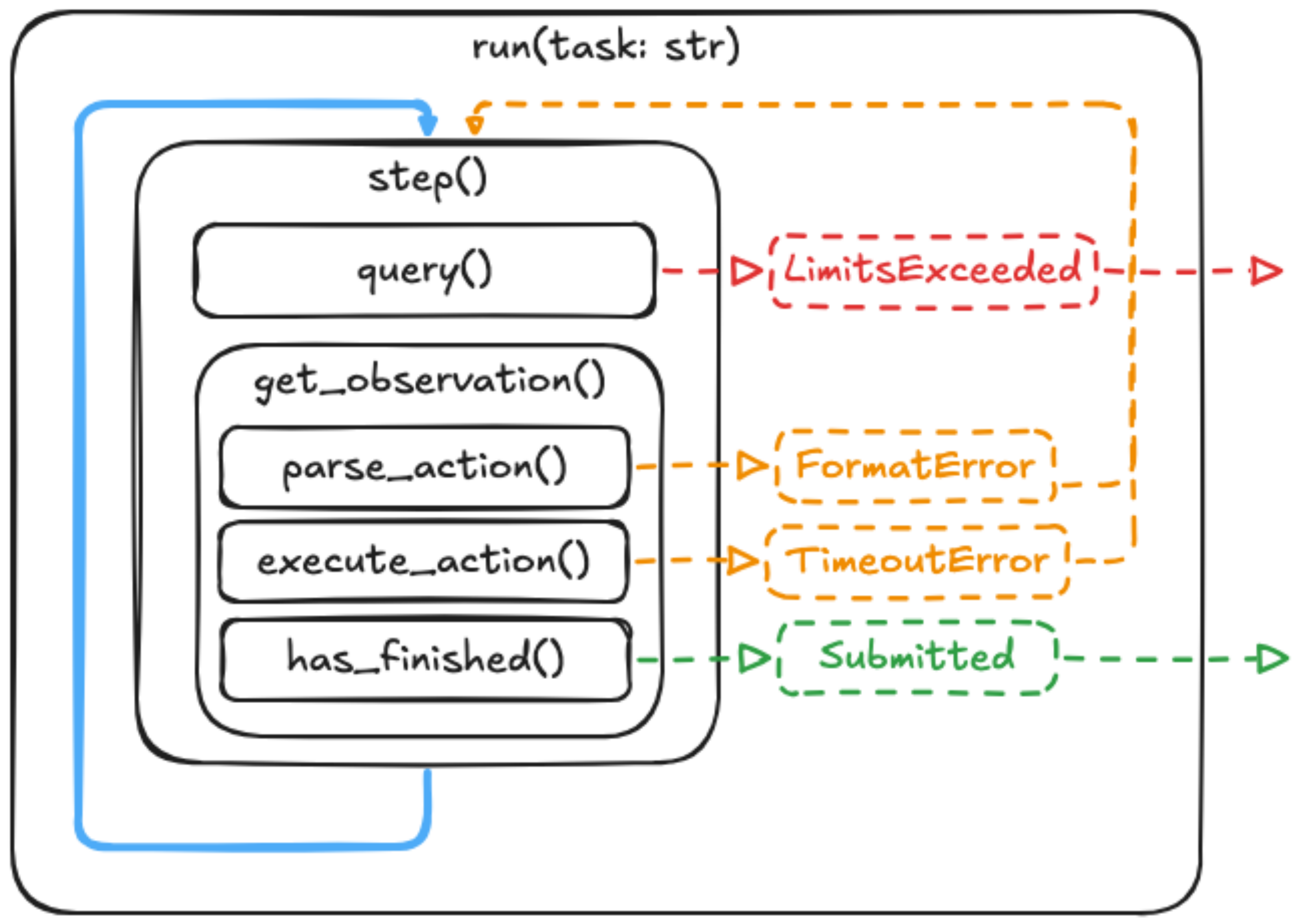
		Microsoft					Other Developers				
		Never	Rarely	Sometimes	Usually	Always	Never	Rarely	Sometimes	Usually	Always
(A)	Phase of the release cycle	2%	6%	17%	35%	37%	14%	11%	27%	22%	16%
	Changes few lines of code	3%	10%	32%	38%	17%	5%	3%	27%	54%	11%
	Requires little testing effort	3%	12%	31%	37%	16%	5%	24%	30%	30%	11%
	Takes little time to implement	3%	10%	43%	30%	13%	3%	14%	35%	30%	19%
(B)	Doesn't change interfaces or break backwards compatibility	0%	2%	8%	36%	53%	0%	0%	14%	32%	54%
(C)	Maintains the integrity of the original design	1%	5%	16%	50%	28%	0%	5%	24%	32%	35%
(D)	Frequency in practice	2%	17%	39%	33%	8%	3%	27%	43%	22%	5%

Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2015. The Design Space of Bug Fixes and How Developers Navigate It. IEEE Trans. Softw. Eng. 41, 1 (Jan. 2015), 65–81. https://doi.org/10.1109/TSE.2014.2357438

Debugging & LLMs

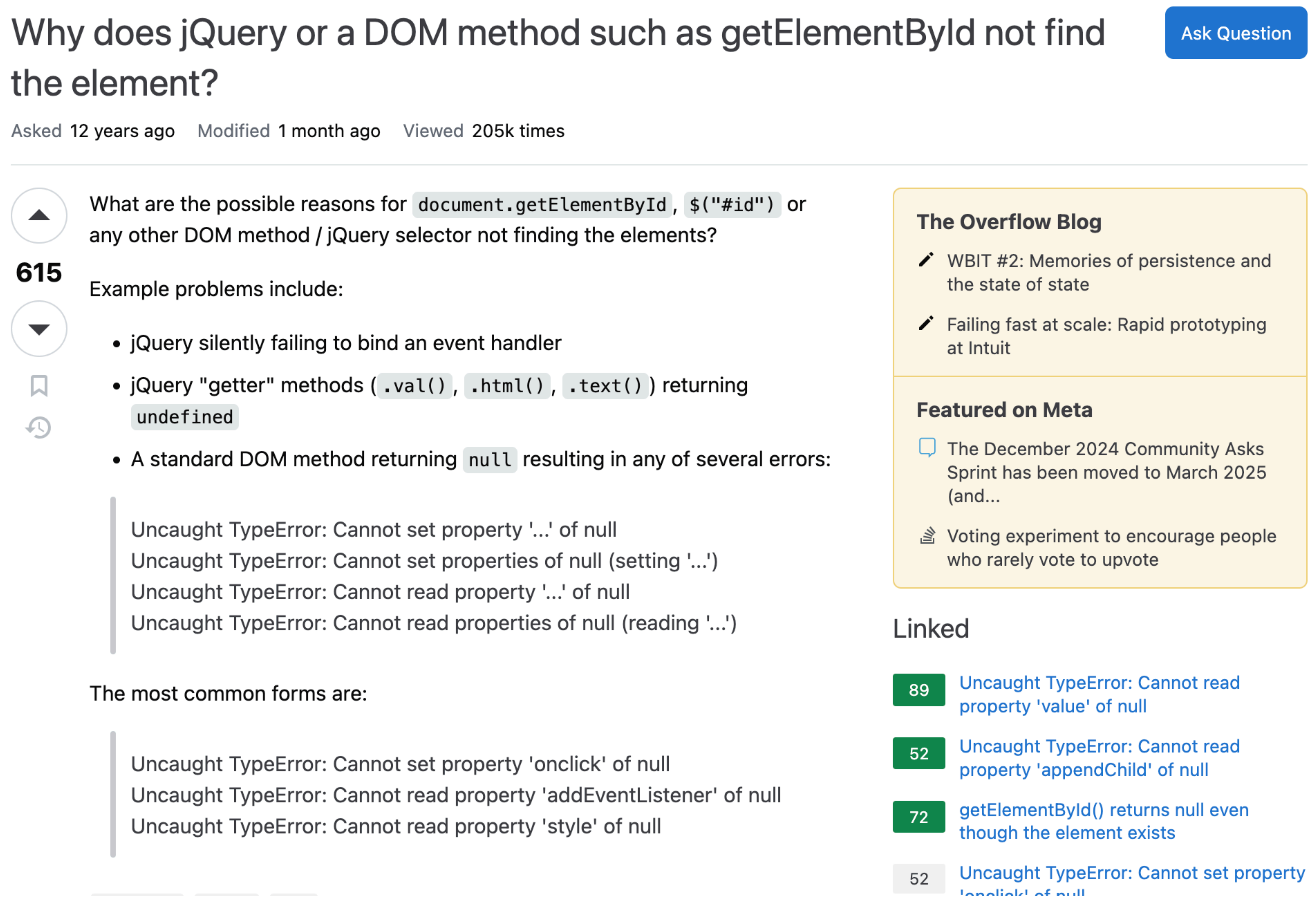
Debugging & LLMs

- Agentic LLMs can use ReACT loop to gather evidence to generate hypotheses and systematically test them
- Pushes key debugging work back to foundation model, responsible for interpreting evidence & generating good hypotheses



Blackbox debugging with an LLM

- LLM has access to StackOverflow data, blog posts, API documentation
- Can use this to interpret error messages or non-working behavior when interacting with a framework


Why does jQuery or a DOM method such as getElementById not find the element?

Asked 12 years ago Modified 1 month ago Viewed 205k times

615

What are the possible reasons for `document.getElementById`, `$("#id")` or any other DOM method / jQuery selector not finding the elements?

Example problems include:

- jQuery silently failing to bind an event handler
- jQuery "getter" methods (`.val()`, `.html()`, `.text()`) returning `undefined`
- A standard DOM method returning `null` resulting in any of several errors:

```
Uncaught TypeError: Cannot set property '...' of null
Uncaught TypeError: Cannot set properties of null (setting '...')
Uncaught TypeError: Cannot read property '...' of null
Uncaught TypeError: Cannot read properties of null (reading '...')
```

The most common forms are:

```
Uncaught TypeError: Cannot set property 'onclick' of null
Uncaught TypeError: Cannot read property 'addEventListener' of null
Uncaught TypeError: Cannot read property 'style' of null
```

The Overflow Blog

- WBIT #2: Memories of persistence and the state of state
- Failing fast at scale: Rapid prototyping at Intuit

Featured on Meta

- The December 2024 Community Asks Sprint has been moved to March 2025 (and...
- Voting experiment to encourage people who rarely vote to upvote

Linked

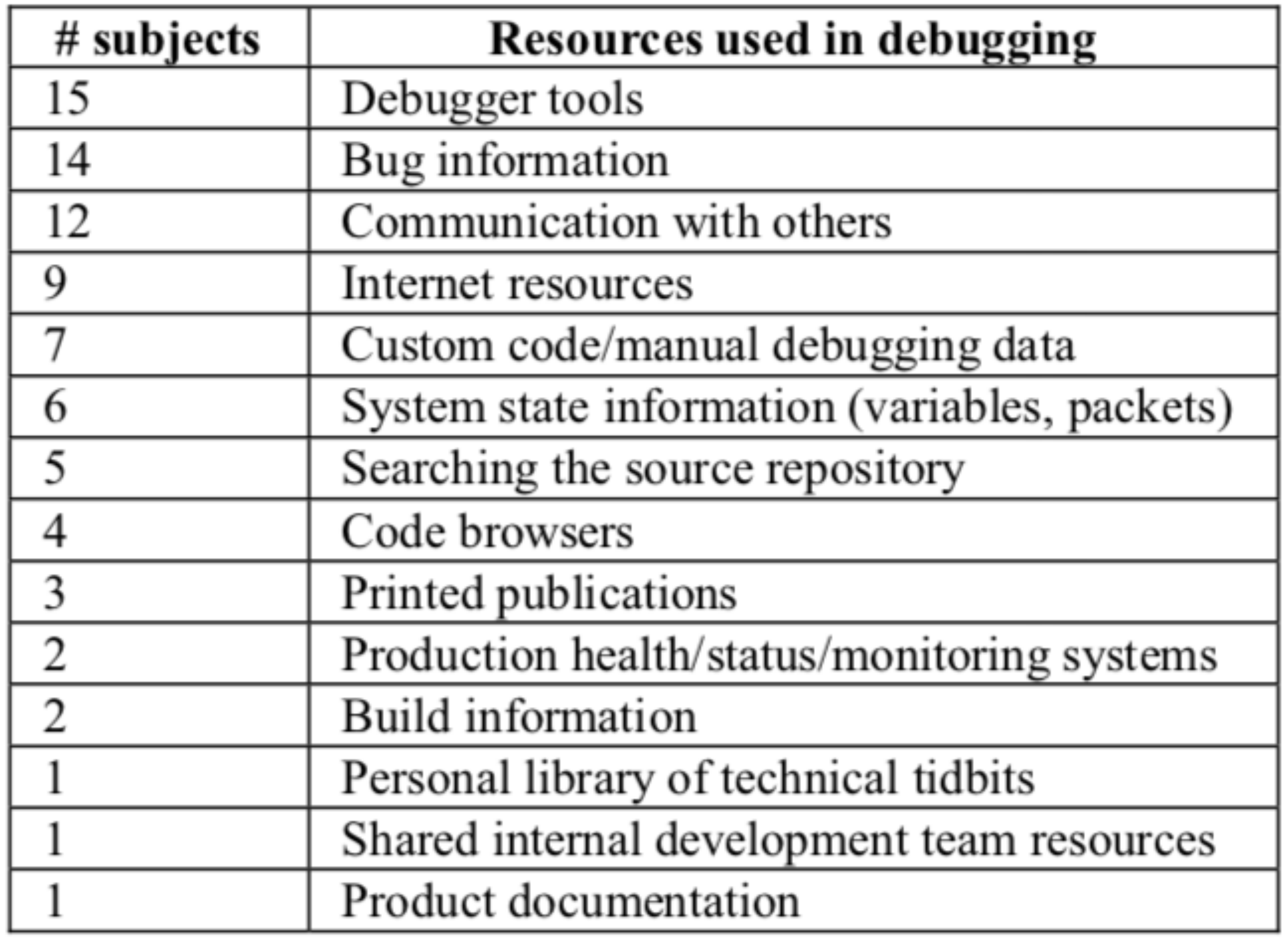
89	Uncaught TypeError: Cannot read property 'value' of null
52	Uncaught TypeError: Cannot read property 'appendChild' of null
72	getElementById() returns null even though the element exists
52	Uncaught TypeError: Cannot set property

Backwards & forward debugging

- LLM can control debugger via command line
- Can use this to trace forwards
- Harder to trace backwards -- needs to repeatedly rerun the program

Hypothesis testing

- Generate a hypothesis, use tools to test
    - Use grep to find relevant code and then read it
    - Add print statements to generate runtime data
    - Edit the code, make changes, see if that works



# subjects	Resources used in debugging
15	Debugger tools
14	Bug information
12	Communication with others
9	Internet resources
7	Custom code/manual debugging data
6	System state information (variables, packets)
5	Searching the source repository
4	Code browsers
3	Printed publications
2	Production health/status/monitoring systems
2	Build information
1	Personal library of technical tidbits
1	Shared internal development team resources
1	Product documentation

Challenges with LLM debugging

- Finding a starting point: in a bigger codebase, where to start? May lack developers' model and understanding of codebase
- Dated API knowledge: foundation model trained on specific version of API, which may no longer be current
- Dated codebase knowledge: LLM can read design doc, but what if design doc is outdated?
- Tool usage steps: LLM may not understand the right command line args necessary to run tools in codebase

- Can try to address by pointing LLM to specific info to add better context

10 min break

In-Class Activity

- In groups of 1 or 2, try to stump an Agentic Debugger (Cursor or Claude Code)
    - Seed bugs in your Code Quality Tool (or another codebase)
    - Goal: insert a defect that breaks something in your City Simulator that agent cannot fix
    - Start with simple defects, try making them more complicated if necessary
    - May need to clear context or reload project so that it can't just debug by looking at recent changes
- Want to hear about what types of defects you tried and why, what worked, and what didn't
    - Ok if can't stump the debugger for all types of defects
- Deliverables
    - Screen recording through Kaltura
        - Upload to OneDrive, turn on link sharing, share link in Lecture 5 activity submission on Canvas
        - Submit answers to questions on your experiences on Canvas (next slide)
- Aim to finish by 7:10pm today;   Due tomorrow at 4:30pm

Types of defects to explore

1. Long cause / effect chain
    1. Can you hide how the output and defect are connected by having lots of code execute in the middle?
2. Black box debugging
    1. Can it debug issues with unpopular framework?
    2. Can it debug issues specific to a recent framework version?
3. Silent failures
    1. Can it figure out where to start when something doesn't happen?

Questions to answer

- What types of defects did you try?
  - Describe how you designed the defect and why you thought it might be hard
- Which of these was the LLM able to debug? Which was it not?
- How did the LLM design fixes to defects?
- What did you learn about when and how LLMs can successfully debug defects?

- **Deliverable**: Submit through Canvas, at least a page