Navigating Complexity: How Context Shapes Debugging Strategy Choices Among Expert Developers

Maryam Arab, School of Information, University of Michigan, Ann Arbor, USA
Jenny T. Liang, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA
Valentina Hong, Department of Computer Science, University of Maryland, College Park, USA
Steve Oney, School of Information, University of Michigan, Ann Arbor, USA
Thomas D. LaToza, Computer Science, George Mason University, Fairfax, USA

***Abstract*—Debugging is a central yet cognitively demanding part of software development, requiring problem-solving skills, expertise, and appropriate tools. Prior work has documented a variety of debugging strategies—such as hypothesis testing, simplification, and forward or backward reasoning—as well as individual factors that influence their use. However, we still lack an integrated understanding of how expert developers select and adapt these strategies in response to changing contextual factors during real-world challenging debugging. In this paper, we conducted a three-phase study: we first synthesized prior work on challenging debugging contexts, then used a short survey of 35 web developers to surface up-to-date examples of difficult defects, and finally conducted semi-structured interviews with 16 expert web developers to understand how they select and adapt strategies in these contexts. We identified a taxonomy of static and dynamic contextual factors that shape strategy selection. Dynamic factors can evolve during debugging and prompt strategy transitions, whereas static factors provide relatively stable constraints on developers' choices. We also present a descriptive state-transition model showing how experts adapt strategies as contextual conditions like clarity, reproducibility, and constraints evolve during a debugging scenario. Our findings highlight the need for more contextual design of debugging tools as well as educational decision-making frameworks for choosing effective strategies.**

***Index Terms*—Debugging strategies, Decision making, Web application defects, Contextual factors, Challenging web application defects.**

I. Introduction

DEBUGGING is a central yet cognitively demanding part of software development. Beyond locating and fixing faults, successful debugging requires developers to interpret symptoms, reason about possible causes, choose appropriate strategies, and adapt their approach as new information emerges [1], [2]. Developers use a range of debugging strategies to structure this reasoning, including hypothesis testing [3], [4], tracing dependencies and execution paths [1], [3], [5], [6], reasoning backward from symptoms to possible causes [3], [4], [7]–[15], and simplifying complex problem spaces [15]–[17].

Such strategies represent higher-level problem-solving knowledge that developers acquire through experience and instruction [18]. Unlike syntax, APIs, or language idioms, this knowledge is often less explicit. Prior work characterizes programming strategies as reusable plans for accomplishing a task—structured sets of actions that guide developers toward a goal [19]. For instance, a strategy might take the form of a reusable procedure for diagnosing a CSS font rendering problem on a website, a systematic way of resolving merge conflicts, or a planned sequence of actions for carrying out a manual refactoring, among many other software development activities [20], [21]. Studies have shown that explicitly documenting and sharing such strategies can improve developer productivity and support learning among less experienced developers [19], [22], [23].

However, debugging strategies are not fixed procedures that developers apply in isolation. Debugging rarely proceeds linearly: symptoms may change, hypotheses may fail, tools may provide incomplete information, or organizational constraints may limit what developers can inspect or modify. Consequently, debugging success depends not only on knowing individual strategies, but also on knowing *when* to apply a strategy and *how* to adapt as circumstances evolve [1], [3], [5], [11], [24], [25]. This adaptive decision-making is often tacit and rarely articulated [19]. As a result, strategies remain difficult to transfer to novices, and tools designed for one strategy (e.g., backward slicing) may be ineffective when a different approach (e.g., defect simplification) is required.

Previous work has studied the mechanics of debugging—which tools are used, how often, and in what ways [26]–[28]. However, far less is understood about developers' decision-making process that guides strategy choice:

Prior studies have examined the mechanics of debugging, including which tools developers use and how they use them [26]–[28]. Less is known about the decision-making process underlying strategy selection and adaptation: the contextual factors developers consider when choosing a strategy, and how those factors lead them to change their approach during a challenging debugging episode.

Bridging this gap is crucial for several reasons. Educating developers on how to select appropriate debugging strategies according to contextual factors can significantly improve their problem-solving capabilities [19], [23]. Moreover, designing tools that adapt to context is important because different strategies may require entirely different debugging tools. Tools designed to aid specific strategies (e.g., slicers for backward

debugging) may become entirely inapplicable in scenarios requiring a different debugging approach (e.g., delta debugging for defect simplification). Finally, a deeper understanding of how developers choose debugging strategies offers researchers more realistic models of software developer behavior.

To investigate this gap, we focus on web development as a rich and challenging domain for studying debugging strategy adaptation. Web applications combine asynchronous and event-driven behavior, interactions across front-end and back-end components, diverse execution environments, browser- and device-specific differences, and rapidly evolving technologies, all of which can complicate debugging [29]–[34]. We focus specifically on challenging scenarios because they make developers' reasoning, tradeoffs, and strategy adaptation more visible. We do not seek to propose a universal model of debugging across all programming domains or to characterize all debugging behavior. Instead, we use challenging web debugging as a focused setting to examine how expert developers choose, adapt, and transition between debugging strategies when standard approaches become insufficient.

We address the following research questions:

RQ1: What types of defects do developers find most challenging to debug in web applications?

RQ2: What context factors do expert developers consider while choosing a debugging strategy in challenging web application defects?

RQ3: How do defect characteristics affect developers' choice of debugging strategies in challenging debugging scenarios?

RQ4: How do codebase characteristics affect developers' choice of debugging strategies in challenging debugging scenarios?

We address these questions through a three-phase study. First, we synthesized prior literature on debugging strategies and challenging debugging contexts to establish an initial strategy vocabulary and set of contextual factors (Phase 0). Second, we conducted a short survey with 35 web developers to identify challenging web debugging scenarios encountered in current practice (Phase 1). Third, we conducted semi-structured interviews with 16 expert web developers to examine how they select, adapt, and transition between strategies in these scenarios (Phase 2).

Our findings show that debugging strategy selection is shaped by multiple contextual factors beyond the defect itself. Some factors, such as tool availability, organizational context, project expectations, and individual habits, tend to remain relatively *stable* during a debugging episode. Others, such as clarity, reproducibility, codebase familiarity, and access, may change as developers gather evidence. These *dynamic* changes often prompt developers to switch strategies, combine approaches, or revise their assumptions.

This paper makes three contributions: (1) a characterization of challenging web debugging scenarios; (2) a taxonomy of contextual factors that influence debugging strategy selection; and (3) a descriptive model of how expert developers adapt and transition between debugging strategies as contextual conditions evolve. Together, these contributions provide a clearer account of adaptive debugging expertise and inform programming education, developer support, and context-aware debugging tools.

II. Related Work

Our work builds on prior research on the activities and challenges of debugging, the strategies developers use when debugging, and the factors that may influence strategy choice.

A. Debugging Activities and Challenges

A long line of work has examined what developers do when they debug and why some bugs are hard to resolve. Classic studies describe debugging as a form of exploratory problem solving: developers ask and answer questions about program behavior [35] and follow a simplified method, forming and testing hypotheses about possible causes [5], [36]–[38].

Researchers have also collected challenges that make debugging more difficult. These include tangled or "spaghetti" codebases, misleading mental models of how a system behaves, and tools that do not match the problem at hand [39]. Work on web debugging and front-end development has identified barriers such as limited tool support for remote or cross-browser debugging, performance issues in multi-threaded or asynchronous applications [24], [27], and insufficient logging for tracing complex client–server interactions [40].

These studies highlight important *obstacles* to successful debugging and describe individual activities (e.g., hypothesis formation, question asking, code navigation). Our work complements this view by asking how expert developers *respond to* such obstacles: which contextual factors they attend to when selecting strategies, and how they adjust those strategies as their understanding of the problem and the codebase evolves.

B. Debugging Strategies and Expert Debugging Practices

Prior research has documented a variety of debugging strategies used by developers. Experts employ well-defined approaches such as hypothesis-testing, systematic code reading, and strategic editing to localize and fix defects [3], [5], [19], [41]. Several studies describe developers constructing and testing theories about failures [3], [13], [42], often combining code inspection with small experiments on the running system [27], [28], [43].

A large body of work distinguishes between *forward* and *backward* reasoning. Forward reasoning proceeds from inputs toward observed behavior, using code navigation and mental simulation to generate facts and hypotheses [1], [3], [5], [6]. Backward reasoning starts from an observed symptom or failure and traces back through execution to identify faulty values or control-flow decisions [1], [4], [7]–[15]. Because traditional debuggers do not support true backward execution [7], researchers have proposed techniques such as program slicing [8] and omniscient debugging to help developers reconstruct past states [7], [9], [10]. Other work describes debugging as unfolding in phases, from initial information gathering and isolation, through hypothesis generation, to deeper investigation and repair [17], [44].

Prior work shows *what* strategies experts use and, in some cases, *how* they apply them. However, it typically treats

strategies in relative isolation, or as a fixed sequence of phases, rather than modeling how experts move *between* strategies in response to changing conditions. Our study builds on these documented strategies but focuses on the *adaptive* aspect: how experts choose among strategies and switch between them as defect and codebase contexts shift during a debugging episode.

C. Factors Influencing Strategy Choice

Several studies have identified factors that influence debugging success. Familiarity with codebase and ability to understand existing code are repeatedly highlighted as central to effective debugging [13], [45], [46]. Proficient debuggers tend to have richer knowledge about program structure and behavior [46], and developers often struggle when working with code written by others or in unfamiliar languages. Spinellis [15] and others advocate tailoring approaches to defect complexity, for example using bottom-up techniques for simple, localized defects and top-down strategies for complex problems involving performance, security, or reliability. Work on faults and fixes has shown that some defects span multiple locations [47], are tied to prior changes [48], or hinge on particular design rationales [49], all of which complicate debugging.

Beyond code and defect properties, work on debugging barriers has pointed to the role of tools, organizational structures, and project constraints. For example, Layman et al. discuss how ownership, process, and communication patterns affect how web developers debug [27], while Evans et al. highlight how time and cost constraints shape how much investigation is considered acceptable in practice [17]. Spinellis' modern debugging guide synthesizes many of these influences and argues that developers must choose appropriate strategies given available tools, time, and risk tolerance [15].

Taken together, this literature identifies many of the *factors* that matter for debugging: codebase familiarity, defect complexity, tool support, and cost constraints, and so on. However, existing work tends to treat these factors separately or as background conditions for success. To our knowledge, there has been little empirical work that (1) develops a *taxonomy* of these contextual factors specifically for challenging web application defects and (2) models how expert developers *adapt their strategy choices over time* as these factors change. Our study addresses this gap by examining how expert web developers reason about context when selecting and switching between debugging strategies, and by proposing a descriptive, context-aware model of expert debugging behavior.

III. Methodology

In our study design, our goal was not to evaluate a specific tool or technique, but to build a contextual account of how experts choose and adapt debugging strategies. We used a three-phase design depicted in Figure 1:

- **Phase 0: Literature synthesis.** We reviewed prior work on debugging strategies and challenging debugging contexts to identify recurring strategy types and canonical "hard problems" (e.g., production failures, distributed systems, concurrency issues).

Fig. 1. We employed a three-phase study design: (Phase 0) literature synthesis to identify debugging contexts and strategies, (Phase 1) scoping surveys to assess the frequencies of challenging scenarios, and (Phase 2) expert interviews grounded in these scenarios to understand strategy selection and adaptation.

- **Phase 1: Scoping survey.** We ran a short survey with 35 web developers to *scope* the landscape of challenging web debugging scenarios in current practice and to validate and update the challenges identified in Phase 0.
- **Phase 2: Expert interviews.** We conducted semi-structured interviews with 16 expert web developers, using literature- and survey-derived scenarios as anchors to elicit detailed accounts of how they select and adapt debugging strategies in context.

A. Phase-0: Literature-Based Preliminary Process of Challenges and Debugging Strategies

Before designing our empirical studies, we conducted a focused literature search to establish a conceptual foundation. Our goals were to: (1) identify debugging strategies repeatedly discussed in prior work, (2) understand which defects and debugging situations prior work describes as challenging, and (3) derive an initial set of contextual factors that might influence strategy choice. Phase 0 synthesized strategies from the broader debugging literature, not only web-development studies. We used this phase to establish a general strategy vocabulary and conceptual foundation; the empirical phases then examined how these strategies were selected, adapted, and transitioned between in challenging web debugging scenarios.

1) Literature Search and Selection: We began with a broad search across the ACM Digital Library, Web of Science, Google Scholar, and IEEE Xplore, focusing primarily on work published between 2000–2023, while including older foundational papers when widely cited. We brainstormed search terms aligned with our research questions, such as debugging challenges, debugging strategies, contextual factors, web development, slicing, backward debugging, hypothesis testing, and forward debugging. We screened titles and abstracts for relevance, then conducted full-text reviews and used backward and forward citation chaining to identify additional studies. In total, we reviewed 43 papers in depth and used 17 as core sources to construct our strategy list and challenging debugging situations.

2) Debugging Strategies Coding Process: To move from paper-level descriptions to higher-level abstractions, we employed an iterative coding process.

a) Identifying Fine-Grained Activities: First, three authors conducted open coding on the descriptions extracted from the 17 core papers to identify recurring debugging activities. This process surfaced specific tactics, including:

- inserting log statements and using print debugging;
- setting breakpoints and inspecting call stacks;
- tracing execution paths or dependencies;
- isolating segments of code or input spaces; and
- generating and testing hypotheses.

b) Synthesizing Macro-Strategies: Authors then grouped these fine-grained activities according to their underlying problem-solving intent. For example, activities such as inserting logs to evaluate an assumption were grouped under *hypothesis testing*. Stepping through execution to identify incorrect program states was grouped under *backward reasoning*. Systematically narrowing the search space by dividing the codebase or input space was grouped under *binary search*.

Next, three authors independently reviewed the grouped activities and synthesized them into higher-level strategy descriptions. This synthesis considered not only the actions themselves but also the conditions under which the literature described them being used. For example, a conditional recommendation such as *"If the program freezes, break the execution by adding breakpoints"* was treated as evidence for when a setting breakpoints tactic supports a broader debugging strategy. The authors then compared their synthesized strategies and refined them through iterative discussion until reaching agreement on the six macro-level strategies in Table I. In the results section, we use "external strategy" to refer to debugging actions that rely on resources outside direct code inspection, such as coordinating with other teams, reproducing user environments, consulting domain experts, or using repository history to identify relevant changes.

3) Derivation of Challenges and Contextual Factors: In parallel with strategy codification, two authors extracted descriptions of challenging debugging situations from the literature and the reason or factor that contributed to the challenge. They focused on identifying recurring environmental and structural barriers that historically complicate the debugging process. These were extracted through a thematic analysis of the "Challenges" or "Barriers" sections or paragraphs of the 17 core papers. The two authors compared their extracted challenges and merged similar ones into higher-level categories. They moved from specific paper-level descriptions to higher-level abstractions by grouping these contexts based on their underlying cause of difficulty. For instance, they identified situations where the deployment setting limits developer actions combined with situations where experimentation and instrumentation are restricted or risky compared to local environments under production-only category. References to multi-component systems, cross-service failures, and massive codebase were grouped under Distributed and Unfamiliar/Huge Code contexts. The output of this analysis was organized into three higher level categories: behaviors, domain, and context of the challenge that provided the initial structure for our study and were integrated into the comprehensive defect landscape in Table III.

TABLE I
SIX DEBUGGING STRATEGIES FROM PRIOR WORK. EACH STRATEGY OUTLINES A SEQUENTIAL APPROACH FOR TROUBLESHOOTING AND RESOLVING DEFECTS. THE COMPLETE DEFINITION OF THE STRATEGIES CAN BE FOUND IN THE SUPPLEMENTAL MATERIALS.

Strategy	Description	Ref
Hypothesis-testing	Forming and testing focused theories via small, testable questions.	[1], [3], [4]
Backward-reasoning	Tracing backward from a failure point to locate incorrect states.	[3], [4], [7]–[14]
Forward-reasoning	Analyzing behavior forward from execution start.	[1], [3], [5], [6]
Simplification	Reducing code/data to isolate minimal failure conditions.	[15]–[17]
Error-messaging	Interpreting and tracing compiler/runtime messages to their source.	[15], [50]
Binary-search	Systematically dividing segments of code or input to isolate defects.	[15], [17]

4) Output: Phase 0 produced three intermediate artifacts: six macro-level debugging strategies (Table I), a set of canonical challenging problems reported in prior work, and an initial set of contextual factors contributing to those challenges. Together, these artifacts provided a conceptual foundation for the Phase 2 interviews and informed the integrated defect landscape in Table III, where literature-derived findings are combined with survey-derived findings from Phase 1.

B. Phase-1: Scoping Survey of Challenging Defects

Phase-1 was designed as a *scoping* step to inform the interview study, rather than as a standalone empirical contribution. We sought a grounded sense of which defects developers currently find most challenging and to validate and update the "hard" debugging contexts identified in Phase 0. Specifically, we aimed to: (1) check whether the literature-derived defect classes reflected current web practice, (2) sample a diverse set of realistic scenarios to use when probing strategy choices in Phase 2, and (3) get a rough sense of which challenges appeared most frequently.

1) Participants: Because Phase 1 aimed to map the landscape of challenging web debugging scenarios rather than analyze expert strategy use, we included developers with a range of experience levels. This helped us surface recurring problems in real-world web systems and ensured that the scenarios later used in expert interviews reflected current practice. The expert-focused analysis of strategy selection and adaptation is reported in Phase 2 (Section III-C).

We recruited participants from advanced graduate-level programming courses that attract students with varied industrial and research experience. We emailed 42 enrolled students with an invitation and a screening survey about their professional experience, web-development background, and familiarity with web frameworks. Inclusion criteria required that participants be at least 18 years old, familiar with web technologies, and have contributed to at least one professional programming project. Of these, 35 met the inclusion criteria and provided sufficiently complete responses for analysis.

Participants reported roles including software developer (n = 11), full-stack developer (n = 7), software researcher (n = 7), software engineer (n = 5), student (n = 2), technical lead (n = 1), and not specified (n = 2). They reported 1–17 years of professional development experience (median: 4) and experience with technologies including Java, JavaScript, Angular, PHP, Python, HTML, CSS, Django, jQuery, JSP, and Go. All participants provided informed consent via email.

2) Survey Design: We conducted a 15-minute online survey using Google Forms. At the beginning of the survey, we provided examples of the *challenging web problems* and clarified the scope of the study. Specifically, we explained that we were not asking about broad, high-level challenges such as overall scalability, long-term maintainability, or architectural security concerns. Instead, we asked participants to describe concrete, recent debugging problems from their day-to-day work.

Participants were asked to recall the most challenging web problem they recently faced and to describe it as if explaining it to another developer who was willing to help. They need to describe: (1) their main goals, (2) the scenario in which the problem occurred (e.g., environment, user actions, relevant components), and (3) the characteristics of the problem that made troubleshooting difficult. To encourage clear, self-contained responses, we asked them to polish their description as if writing a short blog post for a general developer audience.

Following best practices for human-subjects studies [51], we piloted the survey with three software developers. They reviewed early drafts, identified unclear wording, and suggested improvements. We refined the survey iteratively after each round of feedback.

3) Survey Analysis: We treated the survey responses as short narratives. Our analysis was informed by Hammer and Berland's framework for qualitative rigor [52], which treats coder disagreements as opportunities to refine interpretation rather than relying solely on inter-rater statistics. Our goal was to achieve conceptual clarity and consensus through iterative discussion and refinement.

We compiled all responses in a single dataset and focused our qualitative analysis on two dimensions: (1) the *domain or context* of the defect, and (2) the *failure behavior* that participants perceived as contributing to difficulty. For example participant #1 described: *"There was performance concern for calling the API too many times to query data from the database."* We coded this response as both data-related in context and slow performance in behavior. When participants described the underlying technical source, we also coded the domain, such as API or concurrency.

The first two authors independently coded the responses and generated initial descriptive codes for these dimensions. They then compared codes, identified overlaps, merged similar codes, and resolved disagreements through discussion. For example, responses describing 'random failures', 'only sometimes reproducible', or 'hard to trigger consistently' were merged under the category of non-replicable failures. Most disagreements concerned scope or granularity, such as whether to treat "inconsistencies" as a broad category or as a narrower subtype such as "no clear reproduction path." These discussions led to a refined shared codebook. We then counted the number of participants contributing to each category; for example, 14 of 35 participants reported non-replicable failures.

Through this process, we developed a categorized list of defect types that participants reported as difficult to debug. We then combined these categories with the literature-derived contexts from Phase-0 to form the integrated landscape of challenging defects in Table III. The "Source" column in Table III indicates whether each category originated from the literature, the survey, or both, making the contribution of each data source explicit. We discuss these results in Section IV-A.

C. Phase-2: Expert Interviews on Strategy Adaptation

To gain deeper insight into how contextual factors shape debugging strategy choice, we conducted semi-structured interviews with 16 expert web developers (8–38 years of professional experience, all with mentoring backgrounds). Phase 2 examined how experts *reason about*, *select*, and *adapt* strategies while working through challenging real-world defects. The contextual factor taxonomy and the state-transition model reported later in results are grounded in the interviews, the strategies from phase 0 and the problems from phase 1. This study was conducted in 2023, providing a baseline of expert debugging strategies prior to the widespread integration of large language model (LLM) assistants.

We conducted 90-minute interviews via online conference calls, typically split into two sessions. Each interview combined structured prompts (based on the integrated landscape of challenging defects from Table III) with open-ended reflection on recent debugging episodes. All interviews were audio-recorded, transcribed verbatim, and deleted after transcription. Participants received a $50 gift card. Our study was approved by the Institutional Review Boards of our universities. The study instruments including the literature review, the survey and the interview protocol and materials are available in our supplemental materials [53] and the companion website [1].

1) Recruitment: Prior work suggests that mentoring experience is associated with the articulation and development of effective strategies [19], [54], as mentors regularly explain their decisions to learners with varying levels of knowledge. Building on this, we recruited expert web developers with: (1) at least eight years of professional experience in web development and (2) at least three years of mentoring experience (e.g., as team leads, senior engineers, or technical managers).

We recruited through professional networks, emails, snowball sampling, and web development meetups. Recruitment materials described the study goals, time commitment, and eligibility criteria and included a short demographic screening survey. From 54 initial responses, 31 candidates met the inclusion criteria. We then purposefully sampled 16 participants to achieve diversity in roles, organizational contexts, and technology stacks. Recruitment and interviewing continued until thematic saturation was reached, at which point no substantially new contextual factors or strategy-transition patterns were emerging. The remaining eligible candidates were not invited for interviews.

[1] https://sites.google.com/view/navigating-complexity1/

TABLE II
INTERVIEW PARTICIPANT DEMOGRAPHICS. "SOFTWARE ENGINEER" AND "WEB DEVELOPER" ARE ABBREVIATED AS "SWE", "WD".

ID	Job title	Projects (#)	Mentoring (years)	Experience (years)	Technologies/Expertise
P1	WD	9	3	8	Web front-end technologies, scripting, developer tools
P2	Manager	10	4	10	Full-stack development, development tools
P3	SWE	10	5	12	Cloud infrastructure, back-end, developer platforms
P4	WD	48	5	8	IDEs, version control, build tools, debugging utilities
P5	WD	50+	3	8	IDEs, version control, build tools, debugging utilities
P6	Senior WD	70+	3	13	Monitoring/logging, performance metrics, DevOps, testing
P7	SWE (Retired)	7	5	38	Back-end development, data storage, frameworks
P8	Senior SWE	30	5	10	Web frameworks, front-end libraries, component-based UIs
P9	Manager	43	10	19	Web development, UI, typed languages, team leadership
P10	Senior SWE	20+	4	8	Testing, mobile/web apps, CMS, domain-specific apps
P11	WD	17	3	9	JavaScript frameworks, API development, server-side scripting
P12	Principal SWE	20	3	8	Project leadership, software architecture, code review
P13	Front-end Engineer	45+	7	14	Content management, library development, web performance
P14	Senior WD	150+	3	8	Web content management systems, analytics, e-commerce
P15	SWE	8	4	8	Back-end, databases, front-end, AR, DevOps
P16	SWE	20	30	35	Web application development, scripting, system design

2) Participants: Participants were Senior Developer, Principal Engineer, Front-end Web Developer, Software Engineer, and Manager, with professional experience ranging from 8 to 38 years (median: 9.5). They contributed to between 7 and more than 150 projects (median: 20). Participants reported extensive experience with both front-end and back-end technologies, including HTML, CSS, JavaScript, modern web frameworks, server-side languages, and database systems. More details about their experience are provided in Table II.

3) Piloting: To refine our interview protocol and reduce potential bias, we piloted the interview with four software developers, following best practices for human-participant studies [51]. After each pilot, we adjusted question wording, ordering, and the examples used to illustrate key concepts. Our goals were to ensure that (1) the notions of "debugging strategy" and "context factor" were clear and (2) developers could recall and narrate concrete debugging episodes, including moments when they changed strategies and why. Data from these pilot interviews were not included in the final analysis.

4) Design: We split the 90-minute interview into two sessions to reduce participant fatigue and to separate two analytically distinct goals. The first session used structured scenarios derived from Phase 1 to elicit comparable reasoning across participants about common challenging defect types. The second session focused on participants' own recent debugging episodes, allowing us to capture richer narratives of strategy selection and transitions over time.

Before starting, we introduced the study goals and clarified key concepts. We defined *programming strategies* as reusable, high-level plans for accomplishing a debugging task and provided one example (a step-by-step debugging strategy for locating faults) to illustrate the desired level of abstraction. We then shared a Google Doc listing six explicit debugging strategies synthesized from our literature review (Table I) and briefly reviewed each one to establish a shared vocabulary. We emphasized that these strategies were only a starting point and that participants were free to rename, refine, or add strategies based on their own practice. We also introduced the *context factors* as elements or circumstances that can influence the choice of a debugging strategy with examples [17].

a) Session 1: Reasoning about common web defects: The first session lasted 30 minutes and focused on how participants would approach two defect types that emerged as the most frequently reported challenges in Phase 1: non-replicable failures and concurrency-related operations (Table III, underlined). Each scenario was described and presented in a shared Google Sheet with a concrete example.

For each scenario, we asked participants to: (1) outline the initial strategy or combination of strategies they would use, (2) describe strategies they would likely switch to if the initial approach failed, and (3) identify contextual factors that would influence these choices. We then asked follow-up questions about which strategies would be less effective and why, to elicit explicit reasoning about both selection and rejection.

b) Session 2: Recent real-world debugging episodes: The second session grounded the discussion in participants' own recent practice and captured strategy transitions over time. Drawing on the integrated landscape of challenging defects (Table III), we prepared a list of ten complex web problems that were not discussed in Session 1 [5], [27], [45]. The participant selected three problems from this list that resembled defects they had recently encountered.

For each interview, we asked participants to select three problems that closely resembled defects they had encountered in their own work. For each selected problem, we asked participants to walk us step-by-step through how they approached debugging it, from their initial strategy to the eventual resolution. As they narrated, we probed which contextual factors influenced each major decision point, when and why they switched from one debugging strategy to another, and which strategies they had considered but decided not to use.

We also invited comparison across cases, asking why a strategy that worked for a previous problem would be less effective for the current one, and occasionally contrasting their reasoning with earlier participants' accounts (e.g., another developer chose strategy X under a similar factor—would that make sense here, and why or why not?) to probe boundaries and disagreements. This structure encouraged participants to recount concrete debugging episodes rather than abstract preferences and to explicitly connect contextual factors, strategy choices, and strategy transitions over time. These narratives

formed the primary data for our contextual factor taxonomy and state-transition model.

5) Data Analysis: All interviews were audio-recorded, transcribed verbatim, and segmented into analytically manageable units that captured coherent pieces of reasoning about a defect, a strategy, or a contextual factor. Segments were chosen to balance conciseness with contextual richness while avoiding unnecessary fragmentation. Following Hammer and Berland's guidelines for qualitative rigor [52], we used an iterative analysis process that combined deductive codes from phase 0 with inductive codes that emerged from the interviews.

a) Identifying Contextual Factors (RQ2): We began with an initial codebook of contextual factors derived from phase 0 and prior accounts of debugging practices [1], [15], [17]. Three authors independently open coded subsets of transcripts to identify both instances of initial factors and new factors emerging from the data. The authors compared codes, discussed disagreements, refined definitions, merged overlapping concepts, and updated the codebook. Then, they applied the refined codebook to the remaining transcripts, allowing new factor codes to be added when needed. This stage produced the contextual factors reported in Section IV-B, along with representative excerpts across participants.

b) Linking factors to strategies (RQ3&4): To examine how context influenced strategy choice and adaptation, we conducted a second round of coding focused on relationships between contextual factors and debugging strategies. Using causation coding [55], we extracted participants' explicit and implicit statements about how specific contextual conditions shaped their debugging decisions. For each relevant excerpt, we coded: (a) the contextual factor influencing debugging, (b) the strategy selected, discouraged, combined, or revised in response to that factor, and (c) any transition point where the participant described moving from one strategy to another.

For example, P6 described a cross-system inconsistency in which data appeared in one system but not another: *"we see some inconsistencies, like some data shows up in one system, but not in the other ... in such cases, I would try to look into the code. I may not see, like an error message specifically because there was no error, it's just that it did not sync for some reason between systems. I would again go with something like backtracking debugging to see why or where exactly it fails. Now, probably some hypotheses as well. Because if it did not sync with one system, I might have some assumptions or hypotheses about why this could have happened in what scenarios and all."* We coded this excerpt as:

- **Contextual factor:** low observability / lack of explicit error signal, as the failure produced no clear error message and appeared as an inconsistency across systems;
- **Initial strategy:** backward reasoning, because the participant described backtracking through the code to identify where the synchronization failed;
- **Strategy adaptation:** hypothesis-testing, because P6 described forming assumptions about possible scenarios that could explain the synchronization failure;
- **Transition:** backward reasoning $\rightarrow$ hypothesis testing.

We systematically extracted factor–strategy–transition relationships across all interviews and aggregated recurring relationships to identify common patterns of strategy selection and adaptation. These relationships form the basis of Table VII and the state-transition model in Figure 2.

c) Taxonomy construction and model synthesis: After coding was complete, two authors reviewed the evolving codebook and grouped related factor codes into higher-level categories, distinguishing between dynamic and static factors and between defect-, codebase-, environment-, and developer-related factors. We then conducted cross-case thematic analysis, comparing patterns across developers and debugging episodes rather than treating each interview in isolation. We used analytic memos and summary tables to map (factor category, strategy, transition) triples and to identify common patterns and notable exceptions. This iterative synthesis produced (1) the contextual factor taxonomy presented in Sections IV-B and (2) the state-transition model of debugging strategy presented in Section V.

Assessing evidence strength. To improve transparency and interpretability of the transition model, we tracked which participants described each contextual relationship and strategy transition. During axial and causation coding, we recorded instances where participants explicitly described contextual factors influencing strategy choice or transitions between debugging strategies under changing conditions. These participant counts indicate the breadth of supporting evidence for each relationship and are not intended as statistical estimates of prevalence.

Ensuring rigor and validity. We adopted several practices to enhance the credibility and robustness of our analysis. During the initial coding stages, three authors independently coded a subset of the interview transcripts and met regularly to discuss interpretations, refine code definitions, and revise the codebook. Rather than relying solely on inter-rater agreement statistics, coders treated disagreements as opportunities to clarify constructs and strengthen conceptual consistency [52].

Throughout the analysis, the coders maintained shared notes documenting code definitions, codebook revisions, emerging themes, and links between excerpts and higher-level constructs. These notes helped trace how individual excerpts contributed to factor categories, strategy relationships, and the final taxonomy and model. To strengthen credibility, three authors triangulated the findings with the literature- and survey-derived defect contexts, noting where interview accounts confirmed, refined, or challenged those earlier categories.

We judged thematic saturation to have been reached after 14 interviews because no new contextual-factor categories or transition patterns emerged. The final two interviews contributed additional examples and supporting evidence for existing themes but did not introduce substantially new concepts.

IV. Results

The results presented in this section build on outputs from three phases and contributed complementary evidence.

Phase 0 synthesized an initial vocabulary of six debugging strategies, along with a preliminary set of challenging defects and contextual characteristics reported in prior work. Phase 1 expanded the challenge landscape through survey responses,

TABLE III
INTEGRATED LANDSCAPE OF CHALLENGING WEB DEBUGGING PROBLEMS, SYNTHESIZING LITERATURE-DERIVED CONTEXTS AND SURVEY-DERIVED SCENARIOS. WE ORGANIZE CHALLENGES INTO THREE DIMENSIONS: *behaviors*, *domains*, AND *contexts*. THE TWO MOST FREQUENTLY REPORTED SURVEY CHALLENGES ARE UNDERLINED AND WERE USED AS ANCHOR SCENARIOS IN THE FIRST INTERVIEW SESSION.

Challenge Type	Description	Source
BEHAVIORS		
Non-replicable failures	Failures observed at least once, such as in production logs or user reports, but without a reliable way to trigger them again. These failures make it difficult to collect evidence, test hypotheses, or confirm fixes.	Survey (14)
Inconsistent outcomes	Failures where the same input, configuration, or user flow can be repeated, but the outcome alternates between success and failure, suggesting hidden state, timing, or nondeterminism.	Survey (3)
Slow performance	Failures involving excessive resource usage, delayed responses, long loading times, or slow rendering.	Both (2)
DOMAIN		
Concurrency-related	Timing- or order-dependent defects involving asynchronous callbacks, promises, event-driven logic, or concurrent execution, such as races, deadlocks, or synchronization errors.	Both (7)
DOM/UI rendering	Problems in page rendering or user interaction caused by interactions among HTML, CSS, JavaScript, and browser.	Both (6)
External components and APIs	Defects involving third-party components, libraries, APIs, or services, such as incorrect API calls, unexpected data exchange, or unclear or outdated documentation.	Both (4)
Data-flow problems	Incorrect, missing, or stale data transfer between components in stateful systems, such as front-end state management, back-end pipelines, or distributed data paths.	Survey (2)
CONTEXT		
Production-only	Defects that manifest primarily under production workloads or deployment-specific conditions, where direct access, instrumentation, experimentation, or rollback may be constrained or risky.	Literature
Distributed requests	Failures involving actions processed across multiple services, instances, or machines, with difficult control flow.	Literature
Compiler-optimized code	Failures in optimized or transformed code whose runtime structure no longer clearly corresponds to the source.	Literature
Unfamiliar or huge code	Defects in large, complex, or poorly understood codebases (often owned by another team), where navigation, comprehension, and impact assessment are difficult.	Literature
Legacy systems	Defects in outdated systems with limited documentation and historical design decisions that are no longer well understood, increasing the effort required to understand and safely modify behavior.	Literature

identifying defects that developers currently find difficult in practice, including non-replicable failures, inconsistent outcomes, and data-flow problems. Phase 2 then provided detailed accounts of how contextual factors shaped expert developers' strategy selection, adaptation, and transitions during challenging debugging scenarios. Thus, although the final taxonomy and transition model were synthesized primarily from the Phase 2 interviews, all three phases contributed to the final results: Phase 0 provided the conceptual strategies vocabulary, Phase 1 provided empirically grounded challenging scenarios, and Phase 2 provided the evidence for contextual factors and strategy transitions.

A. RQ1: What types of defects do developers find most challenging to debug in web applications?

During coding, we observed that participants described challenging defects along three related dimensions: (1) how the failure appeared, which we call *behavior*; (2) where the likely technical cause was located, which we call *domain*; and (3) the environmental or codebase conditions that made diagnosis more difficult, which we call *context*. We therefore organized the challenge landscape around these three dimensions.

1. BEHAVIORS (failure behaviors) describes *how* the defect manifests to the developer. This dimension includes non-replicable failures, inconsistent outcomes, and slow performance. Non-replicable failures cannot be reliably triggered; inconsistent outcomes occur when the same observable conditions sometimes lead to different results; and slow performance includes delayed responses, long loading times, or excessive resource usage. These behaviors may arise from many different technical domains.

2. DOMAIN (defect domains) describes *where* the root cause tends to reside technically, such as concurrency logic, DOM/UI rendering, API boundaries, or data-flow/state management. These represent architectural or technical defects.

3. CONTEXT (constraints and conditions) describes environmental, codebase, or ownership conditions that make debugging harder but are not themselves the defect. Examples include production-only failures, distributed services, compiler-optimized code, and unfamiliar or legacy codebases.

In Phase 1, participants most frequently reported non-replicable failures (14 of 35 participants) and concurrency- or timing-related problems (7). Other recurring challenges included DOM/UI rendering problems (6) and API or integration errors (4). Fewer participants described inconsistent outcomes or cross-browser/device responsiveness issues (3), state-related data-flow problems in complex client–server or front-end state-management systems (2), and performance-related problems (2).

B. RQ2: What context factors do expert developers consider while choosing a debugging strategy in challenging web application defects?

Our analysis showed that expert developers' strategy choices are shaped by contextual factors that extend beyond the defect itself. We organized these factors into six major categories (Tables IV–VI). Two categories are *dynamic*: (1) **defect characteristics** and (2) **codebase characteristics**. These factors can change or become clearer during a debugging episode as developers gather more evidence. For example, a defect may become reproducible, a failure may become more observable, or developers may discover that the relevant code is part of a legacy subsystem. Such changes often prompt developers to adapt or switch strategies.

The remaining four categories are relatively *static*: (3) **organizational context**, (4) **tool availability and usability**,

(5) **individual developer traits**, and (6) **project expectations**. These factors are shaped by team structures, organizational processes, available infrastructure, and developers' prior habits or preferences. They influence which strategies developers initially consider and which strategies are feasible, but they typically do not change substantially in a debugging episode.

Tables IV–VI present contextual factors identified from the literature review and interview analysis. Some factors naturally overlap because the two sources sometimes highlighted different aspects of the same broader category. We use asterisks to mark factors that originated in the literature-derived codebook and were later refined or extended through the interviews.

1) Defect characteristics (dynamic): Developers described several defect characteristics that directly shape how they approach debugging (Table IV):

Clarity captures how clearly a failure reveals its cause, location, and status as a defect. High clarity occurs when developers have direct diagnostic signals, such as stack traces, error messages, or logs, that point to a likely failure location. Low clarity occurs when signals are missing, misleading, or difficult to interpret, such as when code paths are highly entangled, the behavior only appears in specific environments, or developers are unsure whether the observed behavior is a bug or intended functionality. Thus, clarity includes observability, code-path complexity, ambiguity about intended behavior, and environment-specific manifestation.

Reproducibility captures whether developers can reliably trigger the defect and identify the occurring conditions. While clarity refers to the availability of signals (e.g., logs, error messages, metrics) that help locate or reason about a defect, reproducibility refers to the ability to reliably trigger the defect under controlled conditions. For instance, a defect may be observable without being reproducible (e.g., visible in logs but difficult to trigger), or reproducible with limited observability (e.g., consistently failing with little diagnostic information).

Participants described several sources of low reproducibility, including intermittent, where the same steps do not always trigger the failure; user-specific, where the defect depends on particular accounts, roles, permissions, or user state; compatibility-specific, where behavior differs across browsers, devices, platforms, or configurations; and load-dependent manifestations, where failures occur only under particular runtime conditions such as high traffic, memory pressure, crashes, freezes, or slow rendering.

Root cause captures the suspected or actual origin of a defect. Unlike clarity, which concerns how interpretable the failure signal is, or reproducibility, which concerns whether the failure can be reliably triggered, root cause describes where the defect comes from. Participants described root causes involving dependencies such as APIs, third-party services, libraries, or network calls; data conditions such as size, format, schema, integrity, or status; application layers such as the front end, back end, database, or system state; configuration settings such as environment variables, permissions, feature flags, or deployment setup; and hardware or infrastructure constraints such as hosting, server, memory, storage, or cloud conditions.

These dimensions do not define new defect types; they describe how developers perceive and reason about a defect as they work, and thus how they select and adjust strategies. We analyze their impact on strategy choice in Section IV-C.

2) Codebase characteristics (dynamic): Participants also emphasized properties of the codebase context (Table V):

Familiarity: captures the developer's hands-on familiarity with the specific codebase or component, knowledge of language or framework conventions, and prior exposure to similar failure patterns and scenarios. Participants describe how their prior knowledge, hands-on experience, or familiarity with similar systems helped or limited their ability to understand the code, form hypotheses, navigate the system, or choose an effective debugging strategy.

Accessibility covers access to source code, relevant environments (e.g., production vs. local), and supporting resources such as logs, configuration, and test environments.

Testability captures the extent and quality of automated tests and the availability of debugging and monitoring tools.

Complexity and maintenance state describe how modular or tightly coupled components are, how much technical debt or deprecated code exists, and how large and old the system is.

These are not defect types, but properties of the code context in which defects occur, and they influence how quickly developers can understand a defect and which strategies to apply. Their role in strategy choice is examined in Section IV-D.

3) Organizational context (static): Organizational structures—such as ownership boundaries, reward systems, and cost and time constraints—shape what developers are allowed to change and how much deep diagnosis they can justify (Table VI). For example, debugging a third-party API or a service owned by another team often forces developers toward more constrained strategies: *"Sometimes you're dealing with [a] third-party API (...) so you don't have control over that. And [so] you are limited [to] like binary search debugging and [to] remove stuff (P10)."*

Participants also noted that metrics-based incentives (e.g., number of tickets closed) and cost pressures can push them toward quicker but less thorough debugging approaches.

4) Tool availability and usability (static): Tooling (e.g., debuggers, logs, monitoring, version control) constrains which strategies are practical in a given environment (Table VI). Version control helps relate defects to recent changes; logs and monitoring provide execution traces and signals that support hypothesis formation and narrowing down fault locations. When tools are hard to configure or missing (e.g., no debugger in production, limited logging), developers tend to fall back on strategies that do not depend on them.

5) Individual traits (static): Developers' prior experience and habits (e.g., always starting with print statements, or always opening the debugger first) bias their initial strategy choices (Table VI). These traits influence *which* strategy they try first, even when the context would also support alternatives.

6) Project expectations (static): Finally, project-level expectations—such as deadlines, documentation standards, and performance or reliability targets—affect how thorough developers can be and which strategies are acceptable in practice (Table VI). Participants reported that tight timelines often push them toward faster, less exhaustive strategies, increasing the

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3729587

TABLE IV
THE CONTEXT FACTORS RELATED TO **DEFECT CHARACTERISTICS** DEVELOPERS CONSIDER WHEN SELECTING A DEBUGGING STRATEGY. CONTEXT FACTORS IDENTIFIED IN PRIOR WORK ARE STARRED (*).

Factor	Description
Clarity	
*Observability	Logs, errors, traces, or visible symptoms provide useful diagnostic signals [13], [17], [27], [50].
Complexity	Tangled or distributed exec paths obscure the failure location.
Ambiguity	Unsure if the behavior is a bug or intended feature.
*Environment	The failure appears only in particular environments/configs.
Reproducibility	
*Intermittent	Same steps/conditions inconsistently trigger failure [17], [27]
UserSpecific	Depends on account, role, permission, or state.
*Compatibility	Only on specific browsers, devices, platforms, configs.
*Load	Depends on traffic, resource usage, timing, memory pressure, or runtime load [17], [27].
Root Cause	
*Dependencies	In APIs, third-party services, or network.
*Data-related	Tied to data size, format, integrity, or status [13], [56].
*Layer-specific	Originating in a particular layer, such as front end, back end, database, middleware, or system state.
Configuration	Related to system or environment setup and permissions.
Hardware	Triggered by machine, hosting, cloud, server, storage, memory, or infrastructure constraints

TABLE V
THE CONTEXT FACTORS RELATED TO **CODE-BASE CHARACTERISTICS** DEVELOPERS CONSIDER WHEN SELECTING A DEBUGGING STRATEGY. CONTEXT FACTORS IDENTIFIED IN PRIOR WORK ARE STARRED (*).

Factor	Description
Familiarity	
*Language / Framework	Familiarity with programming language, framework, libraries, conventions, or ecosystem used in the project. [13], [45], [46].
Scenario	Past exposure to similar defects, debugging situations, system behaviors, or failure patterns to similar scenarios.
Accessibility	
Codebase	Ability to view, edit, run, or instrument the relevant source code
*Runtime& Resources	Ability to access environments, APIs, tools, devices, configurations, deployments, logs, or production systems needed for diagnosis [15], [17].
Testability	
*Coverage	Breadth of test cases, data, and environments [3], [15], [27].
Tools	Logs, debuggers, and monitoring systems that support more systematic fault isolation.
Complexity & Maintenance	
Structural	Modularity, tangled code, interconnected components.
Maintainability	code style, documentation, consistency, readability.
*Legacy	Deprecated dependencies, accumulated workarounds, brittle fixes.

TABLE VI
ORGANIZATIONAL, TOOL-RELATED, INDIVIDUAL, AND PROJECT-LEVEL CONTEXT FACTORS THAT SHAPE DEBUGGING. FACTORS THAT OVERLAP WITH PRIOR WORK ARE STARRED (*).

Factor	Description
Organizational Context	
*Ownership & Coordination	Team ownership, decision authority, permissions, communication channels, and cross-team dependencies [27].
*Incentives & Constraints	Time, budget, staffing, resource constraints, and reward structures (e.g., tickets closed, incidents resolved, features delivered) that shape debugging priorities [17].
Tool availability and usability	
*Interactive-debugging	Availability and configurability of debuggers for inspecting program state and execution (locally or in staging)
*Change-history	Access to commit history, branches, diffs, and past code states [15], [17].
Runtime-observability	Availability and quality of logs, dashboards, metrics, alerts, and monitoring systems.
Individual developer traits	
*Habits and preferences	Personal tendencies in how a developer approaches problems (e.g., preference for print statements vs. debuggers).
Project expectations	
Expectation	Project-level constraints such as deadlines, release processes, code review practices, and documentation standards that frame what kinds of debugging activities are acceptable.
Baseline	Shared expectations about acceptable system behavior (e.g., performance thresholds, error budgets, UX standards) that help teams recognize when behavior is considered defective.

risk of leaving underlying issues unresolved or introducing new defects.

C. RQ3: How do defect characteristics affect developers' choice of debugging strategies in challenging debugging scenarios?

Building on the defect characteristics in Table IV, we now examine how clarity, reproducibility, and early beliefs about root cause shape experts' *strategy choices and strategy switches* over time. Participants did not treat these characteristics as static labels; they described them as evolving properties of a defect that often triggered changes in strategy. Table VII summarizes when each strategy tends to work well or poorly under different defect conditions, and Figure 2 synthesizes these patterns into a state-transition model of strategy adaptation. In the following, letters in parentheses refer to nodes and transitions in Figure 2.

1) Clarity: Experts highlighted four aspects of clarity that strongly influenced their initial strategy and subsequent adaptations: observability, complexity, uncertainty about intended behavior, and the environment in which the defect appeared.

a) Observability: The clarity of error messages and exceptions was a primary driver of initial strategy choice. When messages were *clear and specific* (a), stack traces or logs often pointed directly to a plausible fault region. In these cases, experts typically began with *error-message* debugging, inspecting the indicated code and attempting targeted fixes. When they also had full access to the relevant code and runtime environment (d), they often transitioned to *backward reasoning*, using a debugger or detailed traces to step from the failure point back to the root cause (e).

When messages were *unclear or generic* (b), participants shifted to *hypothesis-testing*: forming candidate explanations based on symptoms and recent changes, and inserting temporary code (e.g., logging, assertions) to track execution flow. This instrumentation gradually narrowed the suspect region until backward reasoning became viable (e). Generating useful hypotheses in this situation depended heavily on *familiarity*

with the code and its history (1, m); experts who knew the system well could focus their probes, while those in unfamiliar codebases needed to start with a more exploratory logging and inspection approach first to be able to choose a strategy.

b) Complexity: High local complexity often obscured how observed symptoms could arise from the underlying implementation. When the relevant region was conceptually dense but bounded (l), experts typically began with *forward reasoning*: reading and mentally simulating the code from inputs toward the failure. Once they had narrowed the suspect paths (o), they shifted to *simplification* strategies such as temporarily disabling features, reducing configuration, or commenting out non-essential branches to isolate the behavior causing the defect. Simplification was described as carving out a manageable subproblem, but participants stressed that it worked best when the defect was reproducible and the search space had already been narrowed.

c) Intended Feature: Clarity also hinged on whether the observed behavior was actually wrong. Participants described cases where, for example, a delay in updating a user's balance might reflect an intentional consistency model rather than a bug. In such situations, they first sought to clarify intent: consulting documentation, user stories, or product specifications, and, when needed, talking with stakeholders. This collaborative step helped determine whether the appropriate response was debugging, a feature change, or an interface redesign. Only after clarifying intent did they decide whether to proceed with code-level debugging (e.g., hypothesis-testing and backward reasoning) or treat the observation as a requirements or UX question instead.

d) Environment (production vs. development): Finally, participants emphasized that environment (development, staging, production) constrained how much clarity they could obtain (i). In development or fully provisioned test environments, where they had full code and tool access, experts freely attached debuggers, set breakpoints, and used *backward-reasoning* alongside *forward-reasoning* and *hypothesis-testing*.

In production, access was restricted and user impact had to be minimized. Here, developers relied more on logs, metrics, and other indirect signals (i, 6), treating *error-message* analysis and boundary-focused *hypothesis-testing* as primary strategies. They worked to reproduce issues in safer environments before applying more invasive strategies such as *simplification* or deep *backward-reasoning*. Environment thus acted as a gatekeeper on clarity: test environments enabled direct inspection of internal state, whereas production-only failures pushed experts toward observational and mitigation-oriented strategies until a controlled reproduction could be established.

2) Root Cause: Participants' early beliefs about where a defect likely resided—client vs. server, data vs. configuration vs. external dependency—also shaped strategy choice (c).

For visual or UI issues in the browser, experts often began with high-level *hypothesis-testing* and quickly moved to *simplification* (c, 5). Because many UI elements are not safety-critical, they considered it relatively low risk to comment out or temporarily remove pieces of markup, styling, or JavaScript to see which elements were necessary to reproduce the defect: *"It is safer to delete portions of UI code and put them back without breaking things"* (P1).

This local simplification helped them reduce complex pages to a minimal failing case. At the same time, participants noted that HTML and CSS offer limited support for *backward-reasoning*; there are few exceptions that halt execution in front-end, so traditional debuggers and error-message debugging are less effective for purely structural or layout problems.

For server-side defects (Fig. 2-c), trade-offs differed. External dependencies (e.g., APIs, services) required monitoring and inspecting request–response cycles. Data-related issues demanded tracing how data was validated, transformed, and stored. Configuration or hardware problems required examining settings and machine-specific conditions. Removing or simplifying server-side code was often seen as risky because changes could affect multiple endpoints or shared business logic. When stack traces or recent changes suggested a plausible component, experts preferred *backward-reasoning* over aggressive simplification, stepping from the failure point back through relevant calls. When the suspected location was unclear (b), they rely on *hypothesis-testing* and add targeted logging to narrow where requests or data were being mishandled.

The Logging approach is not considered a strategy by itself, rather it is a way developers rely on to narrow down their decision factors. The success of logging approach depended strongly on codebase familiarity and size (m, n). In small or well-understood services, a few well-placed logs sufficed; in large or unfamiliar systems, excessive logging could overwhelm rather than clarify, leading some experts to first apply local *simplification* (e.g., disabling features or integrations) to bound the search space before continuing with *hypothesis-testing* and *backward-reasoning*.

3) Reproducibility: Seven participants emphasized that reproducibility of a defect strongly influenced which strategies were viable and how hard it was to find the root cause.

a) Inconsistent defects.: Inconsistent defects—failures that appear intermittently under poorly understood combinations of inputs, timing, or environment—were widely seen as among the hardest to debug. They often required extensive logging, detailed user reports, and carefully constructed scenarios even to trigger the issue once.

For these problems, experts relied heavily on *hypothesis-testing* (k). Rather than expecting the bug on every run, they formed hypotheses about underlying causes and looked for indirect evidence supporting or refuting those hypotheses across multiple executions: *"It [hypothesis-testing] is a little bit easier to do if the bug is nondeterministic because even if you do not see the bug on a particular run you might still be able to prove or disprove that hypothesis"* (P12).

By contrast, *simplification* was generally seen as less effective for highly inconsistent defects because removing code did not reliably produce clear evidence about whether the cause had been removed. Participants found simplification or binary search useful only after they could identify a specific, consistently failing sequence of inputs or conditions: *"If there is a particular sequence of inputs that got to the bad state, one could think about trying to use binary search debugging"*

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3729587

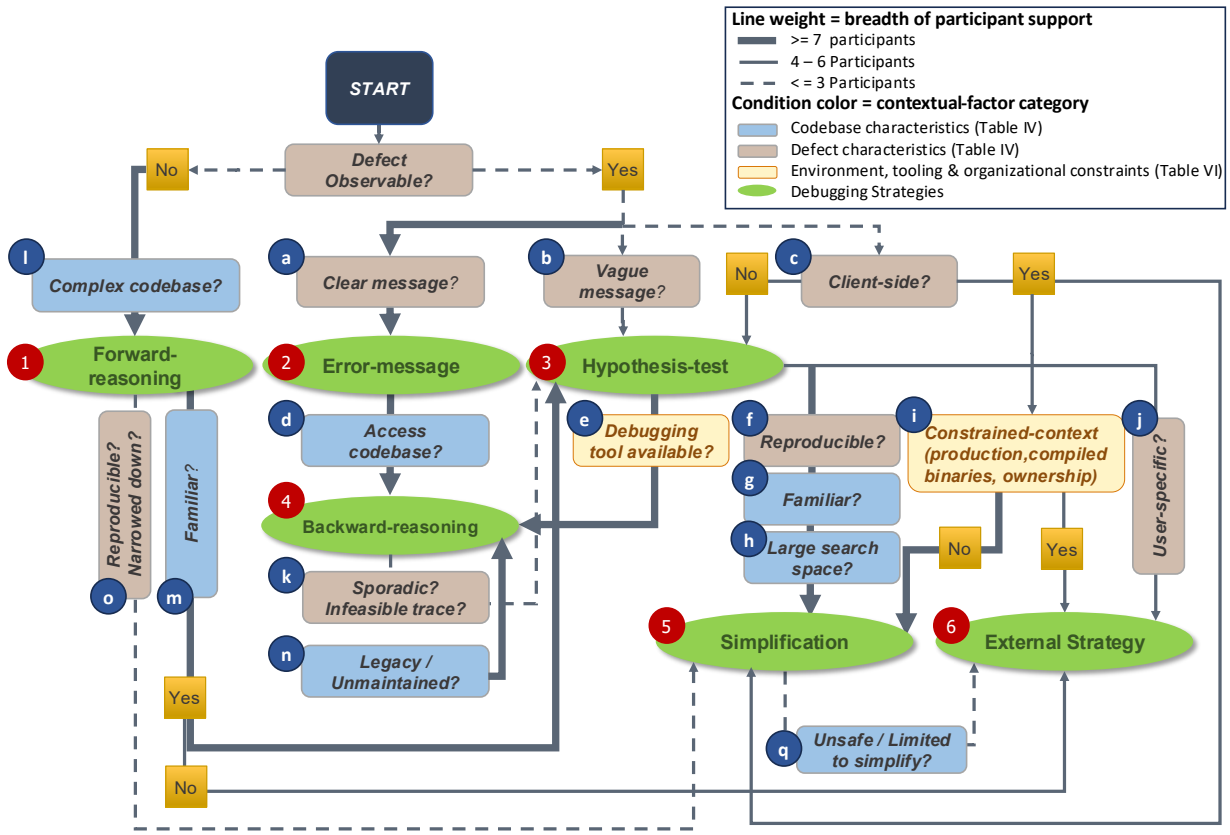

Fig. 2. A summary of commonly reported transition tendencies of strategy adaptation by 16 expert web developers. An absent edge means the transition was not commonly reported, not that it cannot occur. The model is not a decision tree and does not imply that a particular answer to a contextual question determines the next strategy. Debugging strategies are depicted as Nodes (green circles 1-6) represent dominant debugging strategies; directed edges represent common transitions between strategies. Rectangular boxes denote contextual changes that trigger switches. Context factors grouped with color codes. Arrows wight indicates the breadth of supporting evidence, based on the number of participants who described the corresponding transition in Table VII.

(P12). They also noted that many intermittent behaviors were linked to dynamic code paths (e.g., JavaScript or server-side logic) rather than static markup—"HTML is just rendered," as P3 remarked—so they focused on where executable code manipulated state: *"HTML is not executing and it is just rendered, so the inconsistent behavior is related to JS that is executing the code"* (P3).

Although experts often used *backward-reasoning* for server-side defects, 10 experts found it less effective for timing-sensitive concurrency bugs. Breakpoints and added instrumentation could change timing and make the defect disappear ("Heisenbugs"). In these cases, they de-emphasized interactive debugging and instead combined *hypothesis-testing* and *simplification* with lightweight logging and monitoring.

b) User-specific defects.: Participants also described defects that occurred only for particular users or accounts (j), often due to specific roles, preference settings, or historical data. Here, reproducibility depended on faithfully recreating the user's context. Experts relied on *system-level* strategies such as impersonating the user, replicating the defect with the same credentials or roles, and inspecting user-specific configuration and stored data. Only once they could reliably reproduce the behavior under the affected user's settings did more conventional strategies (e.g., hypothesis-testing or simplification) become effective.

D. RQ4: How do codebase characteristics affect developers' choice of debugging strategies in challenging debugging scenarios?

Codebase characteristics (Table V) shape both the strategies developers choose initially and how they adapt as they learn more about a defect. In our data, these characteristics are dynamic: experts often revised their approach once they discovered, for example, that a subsystem was legacy, unusually complex, or hard to access. Below we highlight the roles of familiarity, accessibility, and complexity/maintenance.

1) Familiarity: Eight out of 16 participants described familiarity with the codebase as pivotal for choosing debugging strategies. This aligns with Gould's observation that developers take less time to debug when revisiting a codebase [13], and subsequent research highlighting the importance of program comprehension for successful debugging [5], [45], [57], [58].

When developers understood how components interacted, which modules were fragile, and where similar defects had occurred before, they could quickly prioritize areas to investigate (1, m), generate precise *hypotheses* (3, e), and work *backward* from observed failures to likely causes. They were also more comfortable using *simplification* (m, 5) to carve out a minimal failing case without breaking unrelated functionality. As one expert put it: *"To form a precise hypothesis (...) you have to be very familiar with the code... how the software is put*

TABLE VII
CONTEXTUAL FACTORS THAT INFLUENCE THE EFFECTIVENESS OF DEBUGGING STRATEGIES, DERIVED FROM CROSS-CASE QUALITATIVE ANALYSIS OF 16 EXPERT DEVELOPERS. EVIDENCE INDICATES THE BREADTH OF PARTICIPANT SUPPORT FOR EACH PATTERN. THIS FACTOR-BASED VIEW COMPLEMENTS THE TRANSITION-BASED MODEL IN FIGURE 2.

Strategy	Effective when	Less effective when	Participants Quote
Hypothesis-testing	**Familiarity:** Developer can form plausible hypotheses. (P1, P2, P3, P4, P8, P9, P12, P16) **Reproducibility:** Defect occurs often enough to test assumptions. (P3, P7, P10, P12) **Observability:** Logs or signals support reasoning. (P3, P12)	**Unfamiliar + sparse signals:** Hypotheses become guesswork. (P1, P2, P3, P4, P8, P9, P12, P16) **Slow feedback loops**: Testing hypotheses including builds, deployments, or test runs is costly. (P1, P2, P7, P12)	P10: "if it's consistent and reproducible, then you can immediately move to like making small hypothesis and kind of looking for error messages."
Backward reasoning	**Clarity:** Clear failure point (e.g., stack trace). (P1, P2, P3, P6, P8, P12, P14, P16) **Accessibility:** Debugger and code access available. (P1, P2, P3, P7, P9, P10, P12, P13) **Maintenance:** The relevant code, even if old, is still instrumentable and can be stepped through without excessive risk. (P1, P7, P9, P12, P13, P14)	**Inconsistent:** Breakpoints alter execution in nondeterministic and timing-dependent defects. (P1, P2, P3, P5, P6, P7, P8, P10, P12, P13) **Distributed:** Hard to trace across boundaries. (P12, P15) **Fragile legacy:** Very large, tightly coupled legacy systems make deep tracing prohibitively complex or risky. (P1, P7, P9, P12, P13)	P8: "It's not enough information ... to actually do anything actionable about that. I would need more details. So jumping straight into backtracking is basically going straight into I know where the error is."
Forward reasoning	**Moderate complexity:** Code is mentally traceable. (P1, P2, P3, P4, P6, P9, P10, P13, P14) **Unfamiliar:** Used to build an initial mental model before forming detailed hypotheses. (P1, P2, P3, P8, P12)	**High complexity:** Codebase is very large, tightly coupled, or heavily asynchronous/distributed. (P1, P2, P3, P4, P9) **Low signal:** No clear starting point. (P3, P12)	P1: "The size of the code and the modularity… would help me choose the correct strategy… in all of these cases I use forward debugging."
Error-message	**Clarity:** Clear and specific error messages pointing into a plausible fault region. (P1, P2, P3, P6, P9, P10, P12, P13, P15, P16) **Testability:** Existing tests or reproducible scenarios allow confirming that the error corresponds to the observed failure. (P1, P2, P7, P12)	**Vague or misleading messages:** Errors are generic, swallowed, or point to wrapper layers rather than the true fault. (P1, P2, P3, P4, P6, P9) **UI-only issues:** Pure rendering or layout problems provide little or no actionable error output. (P1, P2, P3, P6)	P12:"we logged something that points to what happened out of order... Error message debugging may not be as useful here, because there may not be an error to debug ... might be some behavior that you know violated the customer's expectations."
Simplification / binary search	**Reproducibility:** Defect consistently occurs. (P3, P6, P7, P10, P12, P13, P15) **Localization:** Narrowed search space. (P1, P2, P3, P6) **Accessibility:** Safe environment for code removal.(e.g., local or test environment, non-critical UI code, good version control and tests). (P1, P3, P6, P10, P13, P14)	**Inconsistent:** Non-reproducible defects do not give clear feedback when code is removed. (P3, P6, P15) **Production-only / constrained access.** (P1, P2, P3, P7, P9, P10, P12, P13) **Fragile legacy systems** may break in unexpected ways when "simplified." (P1, P7, P9, P12, P13, P15	P3: "The first thing I would think about is 'can I repeat it?'… So I would start with either simplification or binary search debugging."
External / environmental strategies	**Constrained access:** Limited ability to inspect code(e.g., limited permissions, production compiled binaries, third-party services). (P1, P3, P6) **High risk:** Need mitigation (e.g., rollbacks, feature flags, traffic shaping) before root cause. (P11, P7) **Ownership boundaries.** (P1, P6)	**Low-risk:** Fully accessible local debugging environments. (P1, P7, P9, P12, P13, P14)	P7: "I think the problem is back deep in the design and the implementation, ... You might be able to get a quick fix by doing something like (simplification), But I'm gonna put a synchronization block on the entire (process)."

together (...) Working backwards from what the problem was and trying to come up with hypotheses that could explain how you got there" (P12). Once they had a plausible fault region, experts often combined *hypothesis-testing* with *simplification* to eliminate irrelevant code and converge on the defect.

In unfamiliar codebases, developers typically started with *forward reasoning* (l, 5) to build a mental model of the architecture and control flow before committing to specific hypotheses. They sometimes anchored this exploration with limited *backward reasoning* (e.g., stepping through a small failing path), and a few mentioned using tools such as ChatGPT to quickly understand unfamiliar languages or patterns in maintenance-heavy code: *"I may not even understand a particular language (...), I have started using ChatGPT where I would just put that piece of code, and it would help me understand what it does"* (P6).

2) Accessibility: Access to code and runtime environments (d, i) also strongly influenced strategy choice (reported by 8 experts) . With full access to source code and environments (e.g., local or staging), developers used more intrusive, fine-grained strategies (d, 4): attaching debuggers, stepping through execution, instrumenting code freely, and reproducing production-like conditions to apply *backward-reasoning*, *simplification*, or *binary-search* over configuration, inputs, or code regions.

By contrast, in *constrained-access* settings (i)—such as production systems with compiled binaries only, strict access controls, or opaque third-party APIs—strategy options narrowed considerably: *"You would pretty much always do this (backward-reasoning) in a local machine (...) You probably cannot do that on production because it's gonna be all compiled binaries"* (P8). In these contexts, developers relied more on external evidence (i, 6): logs, metrics, and monitoring tools to infer behavior at boundaries, and boundary-focused *hypothesis-testing* around integration points. With poorly documented or unsupported APIs, they iteratively adjusted requests and correlated traces across services rather than inspecting internal code. Overall, constrained access pushed debugging toward observational and mitigation-oriented strategies, with fewer opportunities for deep inspection or aggressive simplification.

3) Complexity & Maintenance: Complexity and maintenance state influenced strategy choice in two related ways: overall size/structure and the presence of fragile legacy code.

For *small or moderately sized* codebases, experts often described the code as "mentally traversable." They could read through the relevant region, use *forward-reasoning* to follow expected control flow, and then apply targeted *hypothesis-testing* to confirm or reject candidate explanations.

In *large, structurally complex* codebases, forward-reasoning alone became impractical. Participants relied more on *hypothesis-testing* and *simplification* (i–1, m–3,5): using error messages, logs, and experience to identify a plausible region, then narrowing it via simplification or binary search. Some performed small, opportunistic refactoring to improve local readability, but noted that extensive cleanup was rarely possible under time pressure. Logging remained important but could itself become unwieldy in very large systems. These approaches serve as preliminary steps before developers select a debugging strategy.

Deprecated or fragile legacy code prompted a more conservative strategy profile (n, 6). In old, poorly documented, or minimally maintained subsystems, developers favored *minimal-change* strategies such as *hypothesis-testing* and *backward reasoning* (n, 4), tracing carefully from the failure and making the smallest possible edits to avoid destabilizing other behavior. They viewed aggressive simplification or refactoring as risky in these areas, given unknown dependencies. In some cases, when project constraints allowed, they chose to *replace* clearly obsolete components (e.g., unsupported libraries) as a longer-term maintenance decision, distinct from immediate incident response.

V. Strategy Adaptation State-Transition Model

We synthesized our findings into two complementary views of debugging strategy selection. The first is *factor-based view* (Table VII), summarizes when each strategy tends to be effective or ineffective under different contextual conditions. The second is a *transition-based view* (Figure 2), which captures how developers adapt their strategies over time as those conditions evolve. Together, these views show that expert developers rarely commit to a single strategy for an entire debugging episode. Instead, they adjust their approach as they learn more about the defect, encounter constraints, or narrow the problem space. While a factor-based representation captures when strategies are effective, it does not represent how developers adapt their approach over time. The transition model therefore complements the factor-based view by capturing this temporal and adaptive aspect of debugging behavior.

Figure 2 represents debugging as a sequence of *strategy states* and *transitions*. Each node (green oval) corresponds to a dominant debugging strategy reported by participants, and each directed edge represents an observed switch between strategies. Edge labels (rectangles) summarize contextual changes that participants described as triggering or constraining those switches, such as a defect becoming reproducible, the search space remaining too large, the discovery of a legacy subsystem, or restricted access to production.

The model should *not* be intended as a deterministic decision tree or a prescriptive algorithm. Rather, it is a descriptive model of common strategy-transition patterns observed in our interviews. Developers may follow different paths, skip intermediate strategies, combine strategies, or directly identify a solution without moving through multiple states. For example, developers may begin with forward reasoning to build a mental model of unfamiliar code. If this process reveals that the search space remains large or the cause is unclear (M-No), they may shift to external strategies to explore possible explanations. If the defect later becomes reproducible and the relevant region is narrowed (O), they may move to simplification to isolate the failure. However, forward reasoning may also lead directly to identifying the root cause, in which case no transition is necessary.

Because the model is derived from retrospective interviews with 16 expert web developers, it reflects the most salient and frequently articulated transitions in our dataset rather than all possible debugging paths. Some less common transitions may therefore be absent. Recall bias may also lead participants to emphasize memorable or difficult episodes over routine ones. We therefore interpret the model as a descriptive representation of typical transition tendencies in challenging web debugging, rather than a complete map of debugging behavior. Future work using larger samples or observational studies of real-time debugging could extend the model to capture additional or less frequent strategy paths.

Table VII complements the transition model by summarizing the conditions under which each strategy tends to help or become less effective. While our qualitative approach does not aim to estimate precise frequencies, we include participant counts to indicate the breadth of support for each pattern across interviews.

Hypothesis testing was commonly useful when developers had enough familiarity with the codebase and enough diagnostic signal from logs, error messages, or prior experience to propose plausible explanations. In these cases, they can propose plausible explanations and run a series of small experiments to confirm or rule them out. It became less effective when code was unfamiliar, runtime evidence was limited, or experiments were too slow, risky, or costly to repeat.

Backward-reasoning was most helpful when developers had a clear failure point and sufficient access to trace the system backward from an observed symptom. For example, a crash, incorrect output, or localized failure in a local or staging environment made it possible to inspect relevant states and trace likely causes. It was less effective for timing-sensitive defects, distributed systems, opaque third-party components, or production-only failures where relevant state was difficult to observe or access.

Forward reasoning was useful when developers needed to build an initial understanding of unfamiliar code by following execution from inputs toward observed behavior. Participants found it most effective in small or moderately sized, modular parts of the system. In large, tightly coupled, or heavily asynchronous systems, forward reasoning could become overwhelming and often gave way to more targeted strategies.

Error-message debugging was effective when exceptions, stack traces, compiler messages, or logs were specific and trustworthy enough to point to a plausible fault region. It was less helpful when messages were vague, misleading, hidden

by abstraction layers, or absent, such as in some UI-only rendering problems.

Simplification, including *binary-search* strategies over code, configuration, or inputs, was effective once a defect was reproducible and narrowed to a manageable region. Participants described systematically reducing the problem to a minimal failing case. It was less suitable for timing-sensitive defects, production-only failures where changes were risky, or legacy systems where simplification could introduce new faults.

Finally, *external or environmental strategies*, including refactoring (P11), system-level strategies for mimicking a user role (P3, P6), coordinating with other teams and experts (P6, P1), and using Git repository for isolating the exact commit that introduced a bug using "git-bisect" command (P7, P10), were most prominent when access was constrained, risk was high, or ownership was distributed across teams and domain experts. When a problem could be reproduced safely in a local environment and was confined to code the developer controlled, participants preferred code-focused strategies.

Taken together, these patterns show that expert debugging is adaptive: each strategy has conditions under which it is more or less effective, and experts move between strategies as clarity, reproducibility, familiarity, and constraints change.

VI. Limitations

While our study offers insights into how experts account for contextual factors when debugging, several limitations remain. Our model captures observed reasoning and adaptation patterns under complexity and uncertainty, rather than a universal process for all debugging tasks. Hence, it should be viewed as a first, partial account of context-aware strategy use in expert web debugging, not a complete theory of debugging across all domains. Because this is a qualitative, interview-based study, we assess its limitations against criteria appropriate to qualitative inquiry including credibility, transferability, rigor, transparency, and scope [59]

Credibility. Our analysis relies on self-reported, retrospective accounts of debugging episodes, which are subject to recall bias, incomplete accounts, and subjective interpretation. Participants may emphasize memorable or difficult cases and omit routine or less salient transitions, and despite standardized instructions they may have interpreted "challenging debugging problem" differently, biasing our sample of problems toward extreme cases. We mitigated this by asking for concrete recent episodes rather than abstract preferences, by encouraging participants to consult artifacts such as repositories, commit history, and documentation while narrating, and by probing decision points with follow-up questions; some memory error nonetheless remains likely, and some strategy paths may not have been fully recounted. Because we did not observe live debugging sessions or collect fine-grained interaction logs, our state-transition model reflects how participants *remember* moving between strategies rather than time-aligned traces of behavior. We also operationalized expertise as at least eight years of professional experience together with mentoring responsibility—a pragmatic but imperfect proxy, given the absence of validated measures of debugging proficiency—and, because much prior literature concerns non-expert developers, the scenarios we derived from it in Phase 0 may not span the full breadth of expert practice. Our interview scenarios were likewise bounded by the tasks participants chose to describe.

Transferability. Our findings are bounded by sample and domain. They rest on 16 expert developers with substantial web-development experience reasoning about challenging web defects. Web development has distinctive challenges and tool chains, and our findings may not carry to domains such as embedded systems, operating systems, mobile applications, or data-science pipelines, where debugging constraints, execution models, and observability mechanisms differ substantially and where other contextual factors—hardware limits, nondeterminism in ML models—are central. Hence, the taxonomy and transition model should not be read as a universal account of debugging across programming domains, developer populations, or routine tasks. Developers working in other domains or with different levels of expertise may encounter additional factors or strategy transitions. Future work with larger and more diverse samples could extend the model to capture additional behaviors across domains and developer populations. Several core strategies we identify, such as hypothesis-testing and backward reasoning, are grounded in the broader debugging literature and may remain recognizable in those settings, though their usage patterns and transitions are likely to differ. Our interviews also captured no use of modern AI-assisted debugging tools, and we did not model how large language model copilots or automated log-analysis services fit into experts' strategy repertoires, which may limit applicability in AI-heavy tool ecosystems. Broader domain-spanning studies, including longitudinal and observational work, are needed to validate and extend the model across domains, developer populations, and task types.

Rigor. We used several practices to strengthen the soundness of our analysis: independent coding by multiple authors, iterative refinement of the codebook, analytic discussion to resolve disagreements, cross-case comparison, and triangulation with both the literature and the survey findings. We judged thematic saturation when no new contextual-factor categories or transition patterns emerged; saturation in this sense concerns the stability of categories rather than the enumeration of every possible path through the model. Some interpretation variance is nonetheless inherent in qualitative work. Recruitment through professional networks and snowball sampling, while effective in reaching experts, may overrepresent certain specialties. Finally, our multi-phase design broadened coverage of defect contexts but was not intended to estimate how frequently any factor or transition occurs; the participant counts we report indicate the breadth of supporting accounts, not prevalence.

Transparency. To support scrutiny and reuse, our study instruments—the literature review protocol, the survey, and the interview guide—together with the definitions and examples given to participants, are available in our supplemental materials [53] and on the companion website.

Scope. Our findings are bounded by both sampling and domain. Recruitment through professional networks and snowball sampling, while effective in reaching experts, may overrepresent certain demographics or specialties. The transition

model captures the strategy transitions most clearly articulated in our dataset and is not intended to represent every possible debugging path. Developers may combine strategies, skip intermediate steps, or arrive directly at a solution. Similarly, the absence of a transition from the model should not be interpreted as evidence that such a transition does not occur.

VII. Discussion and Future Implications

Our findings reaffirm that debugging is not just a sequence of tool operations, but a context-sensitive cognitive activity shaped by both dynamic characteristics of defect and codebase and relatively static individual, organizational, and tool constraints. In line with work on situated problem solving and sensemaking in programming [60]–[62], experts in our study continually reinterpret context as they work: as clarity, reproducibility, or access change, they often pivot from one strategy to another. Our contribution should therefore be read as an integrative account that links documented strategies to an explicit vocabulary of context factors and typical strategy transitions, not as a proposal for entirely new techniques.

Although our analysis focused on expert web developers, the framework raises questions for other domains where debugging contexts differ significantly. For example, embedded developers must account for hardware–software interactions, while data scientists face issues of stochasticity and model opacity. Extending our taxonomy across domains could both test its robustness and reveal new categories of contextual influence. Furthermore, our expert sample, developers with 8 to 38 years of experience, many of whom mentor others, suggests that novices may perceive or respond to context differently, perhaps relying more on procedural guidance than adaptive reasoning. Understanding how awareness of context develops over time could inform both professional training and CS education. More broadly, future work should examine how organizational cultures, team structures, and emerging AI-based tools mediate developers' sensitivity to context and their willingness to switch strategies, particularly in high-pressure or safety-critical settings.

Our state-transition model is descriptive rather than prescriptive: it captures common patterns in experts' accounts but is not a strict decision procedure. Its main value lies in surfacing the kinds of contextual cues experts attend to and how these cues tend to influence strategy choice. Our findings have implications for programming learners and practitioners.

For learners, the challenge is often not knowing *a* strategy, but knowing *which* strategy fits the current context. Our findings suggest teaching and mentoring should emphasize context sensitivity and reflective practice. Educational tools could, for example, prompt students to explain why they switched from one approach to another (e.g., from tracing to hypothesis-testing) or ask them to explicitly assess factors such as clarity, reproducibility, and access before committing to a strategy. Mentors could similarly annotate real debugging episodes with short notes about contextual cues (*production-only, legacy, limited access*) and chosen strategies, building small libraries of case studies that make expert reasoning visible. Future work can examine whether explicitly surfacing our factor categories helps learners choose and sequence strategies more effectively, or whether additional scaffolds are needed.

Practitioners can think about designing context-aware debugging tools. Most debugging environments treat problems as static, but our taxonomy suggests tools that respond to changes in context. For example, an IDE could detect that a defect appears intermittent and suggest logging asynchronous events, replaying executions, or isolating nondeterministic behavior. When activity patterns indicate unfamiliar code (e.g., many jumps, frequent backtracking), tools could surface lightweight architectural overviews or dependency graphs. Combining runtime traces, version-control history, and monitoring signals might also help tools infer when a developer's context has shifted (e.g., from local reproduction to production-only incidents) and adjust suggestions accordingly. Evaluating such context-aware tools will likely require a mix of lab studies, field deployments, and analysis of naturally occurring debugging traces to avoid overly intrusive instrumentation.

The Evolving Role of AI in Strategy Selection. Although this study establishes a pre-AI baseline, our State-Transition Model (Figure 2) provides a robust framework for understanding how modern AI assistants accelerate expert debugging. In modern software development practice, these models might primarily serve as catalysts within the decision-making loop:

- Accelerating Hypothesis-Testing (Node 3): LLMs can rapidly generate candidate explanations for vague or misleading errors, potentially shortening the initial evidence-gathering phase.
- Facilitating Simplification (Node 5): By helping developers interpret complex or legacy dependencies, AI lowers the risk threshold for removing code to isolate defects.
- Serving as an External Strategy (Node 6): AI interaction functions similarly to expert consultation, providing rapid architectural overviews when navigating unfamiliar languages or frameworks.

However, these AI tools might introduce new contextual factors, such as AI trust and hallucination risk, which could trigger a transition back to manual Forward-reasoning. Ultimately, while AI reduces the time and effort required for specific transitions, the fundamental cognitive requirement to adapt strategies based on shifting context remains a cornerstone of expertise.

VIII. Conclusion

We have argued that debugging is best understood as a context-sensitive strategic activity shaped by a mix of dynamic (defect and codebase) and static (organizational, tool, individual, project) factors. Expert web developers in our study adaptively navigate these factors, switching strategies as their understanding of the problem and system evolves. By articulating how contextual cues shape strategy selection and transitions, our work provides a conceptual framework for researchers and practical guidance for tool designers and educators. Future studies should extend and challenge this framework across domains, levels of expertise, and real-world settings, with the goal of designing tools, AI copilots, and training practices that better match how programmers actually debug.

REFERENCES

[1] D. Spinellis, *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. Boston, MA: Addison-Wesley Professional, 2016. [Online]. Available: https://www.spinellis.gr/debugging/

[2] ——, *Code reading: the open source perspective*. Boston: Addison-Wesley Professional, 2003.

[3] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 117–128.

[4] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Transactions on software engineering*, vol. 30, no. 12, pp. 889–903, 2004.

[5] I. R. Katz and J. R. Anderson, "Debugging: An analysis of bug-location strategies," *Human-Computer Interaction*, vol. 3, no. 4, pp. 351–399, 1987.

[6] P. Romero, B. Du Boulay, R. Cox, R. Lutz, and S. Bryant, "Debugging strategies and tactics in a multi-representation software environment," *International Journal of Human-Computer Studies*, vol. 65, no. 12, pp. 992–1009, 2007.

[7] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Software: Practice and Experience*, vol. 23, no. 6, pp. 589–616, 1993.

[8] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[9] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 151–158.

[10] B. Lewis and M. Ducassé, "Using events to debug java programs backwards in time," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2003, pp. 96–97.

[11] F. Lukey, "Understanding and debugging programs," *International Journal of Man-Machine Studies*, vol. 12, no. 2, pp. 189–202, 1980.

[12] S. Jiang, C. McMillan, and R. Santelices, "Do programmers do change impact analysis in debugging?" *Empirical Software Engineering*, vol. 22, pp. 631–669, 2017.

[13] J. D. Gould, "Some psychological evidence on how people debug computer programs," *International Journal of Man-Machine Studies*, vol. 7, no. 2, pp. 151–182, 1975.

[14] J. Engblom, "A review of reverse debugging," in *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. IEEE, 2012, pp. 1–6.

[15] D. Spinellis, "Modern debugging: the art of finding a needle in a haystack," *Commun. ACM*, vol. 61, no. 11, p. 124–134, oct 2018. [Online]. Available: https://doi.org/10.1145/3186278

[16] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on software engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[17] J. Evans, *The Pocket Guide to Debugging: Stellar Strategies for Sticky Situtations*. .: Wizard Zines, 2022.

[18] M. de Raadt, R. Watson, and M. A. Toleman, "Chick sexing and novice programmers: explicit instruction of problem solving strategies," in *Australasian Conference on Computing Education (ACE)*, 2006.

[19] T. D. LaToza, M. Arab, D. Loksa, and A. J. Ko, "Explicit programming strategies," *Empirical Software Engineering*, vol. 25, pp. 2416–2449, 2020.

[20] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," in *International Symposium on Software Testing and Analysis*. ACM SIGSOFT, 1996, pp. 121–134.

[21] M. A. Francel and S. Rugaber, "The value of slicing while debugging," *Sci. Comput. Program.*, vol. 40, pp. 151–169, 2001.

[22] M. Arab, J. Liang, Y. Yoo, A. J. Ko, and T. D. LaToza, "Howtoo: A platform for sharing, finding, and using programming strategies," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2021, pp. 1–9.

[23] A. J. Ko, T. D. LaToza, S. Hull, E. A. Ko, W. Kwok, J. Quichocho, H. Akkaraju, and R. Pandit, "Teaching explicit programming strategies to adolescents," in *Proceedings of the 50th ACM technical symposium on computer science education*, 2019, pp. 469–475.

[24] T. D. LaToza and B. A. Myers, "On the importance of understanding the strategies that developers use," in *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, 2010, pp. 72–75.

[25] S.-e.-Z. Haidry, K. Falkner, and C. Szabo, "Identifying domain-specific cognitive strategies for software engineering," in *Conference on Innovation and Technology in Computer Science Education (SIGCSE)*, 2017, pp. 206–211.

[26] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 572–583.

[27] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia, "Debugging revisited: Toward understanding the debugging needs of contemporary software developers," in *2013 ACM/IEEE international symposium on empirical software engineering and measurement*. IEEE, 2013, pp. 383–392.

[28] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," *Software Quality Journal*, vol. 25, pp. 83–110, 2017.

[29] S. Hong, Y. Park, and M. Kim, "Detecting concurrency errors in client-side java script web applications," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 61–70.

[30] J. W. Mickens, J. Elson, and J. Howell, "Mugshot: Deterministic capture and replay for javascript applications." in *NSDI*, vol. 10, 2010, pp. 159–174.

[31] E. Mutlu, S. Tasiran, and B. Livshits, "Detecting javascript races that matter," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 381–392.

[32] Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 805–814.

[33] F. S. Ocariza Jr, K. Pattabiraman, and B. Zorn, "Javascript errors in the wild: An empirical study," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 2011, pp. 100–109.

[34] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side javascript bugs," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 55–64.

[35] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 23–34.

[36] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," *Software Quality Journal*, vol. 25, pp. 83 – 110, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:14856284

[37] J. D. Gould and P. Drongowski, "An exploratory study of computer program debugging," *Human Factors*, vol. 16, no. 3, pp. 258–277, 1974.

[38] M. S. Carver and S. C. Risinger, "Improving children's debugging skills," in *Empirical studies of programmers: Second workshop*, 1987, pp. 147–171.

[39] M. Eisenstadt, "My hairiest bug war stories," *Communications of the ACM*, vol. 40, no. 4, pp. 30–37, 1997.

[40] D. I. Samudio and T. D. LaToza, "Barriers in front-end web development," in *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2022, pp. 1–11.

[41] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[42] M. Eisenstadt, "Tales of debugging from the front lines," in *Empirical Studies of Programmers: Fifth Workshop*. Palo Alto, CA: Ablex Publishing Corporation, 1993, pp. 86–112.

[43] A. Zeller, *Why programs fail: a guide to systematic debugging*. Burlington, MA: Elsevier, 2009.

[44] A. Alaboudi and T. D. LaToza, "What constitutes debugging? an exploratory study of debugging episodes," *Empirical Software Engineering*, vol. 28, no. 5, p. 117, 2023.

[45] M. Ahmadzadeh, D. Elliman, and C. Higgins, "An analysis of patterns of debugging among novice computer science students," in *Annual Conference on Innovation and Technology in Computer Science Education*, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:18292779

[46] M. Decasse and A.-M. Emde, "A review of automated debugging systems: Knowledge, strategies and techniques," in *Proceedings.[1989] 11th International Conference on Software Engineering*. IEEE Computer Society, 1988, pp. 162–163.

[47] L. LUCIA, F. Thung, D. Lo, and L. Jiang, "Are faults localizable?" 2012.

[48] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 55–64.

[49] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design space of bug fixes and how developers navigate it," *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 65–81, 2014.

[50] D. Spinellis, "Debuggers and logging frameworks," *IEEE software*, vol. 23, no. 3, pp. 98–99, 2006.

[51] A. J. Ko, T. D. LaToza, and M. M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, 2015.

[52] D. Hammer and L. K. Berland, "Confusing claims for data: A critique of common practices for presenting qualitative research on learning," *Journal of the Learning Sciences*, vol. 23, no. 1, pp. 37–46, 2014.

[53] Anonymous, "Supplemental materials to "navigating complexity: How context shapes debugging strategy choices among expert developers"," 2023, available at https://doi.org/10.6084/m9.figshare.32760969.

[54] M. Arab, T. D. LaToza, J. Liang, and A. J. Ko, "An exploratory study of sharing strategic programming knowledge," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–15.

[55] S. Patel, "A guide to coding qualitative data," *Retrieved from*, 2014.

[56] D. Spinellis, "Differential debugging," *IEEE Software*, vol. 30, no. 5, pp. 19–21, 2013.

[57] L. Gugerty and G. Olson, "Debugging by skilled and novice programmers," in *Proceedings of the SIGCHI conference on human factors in computing systems*, 1986, pp. 171–174.

[58] M. Nanja and C. R. Cook, "An analysis of the on-line debugging process," in *Empirical studies of programmers: Second workshop*, 1987, pp. 172–184.

[59] R. Hoda, *Qualitative Research with Socio-Technical Grounded Theory: A Practical Guide to Qualitative Data Analysis and Theory Development in the Digital World*. Cham, Switzerland: Springer, 2024.

[60] A. J. Ko, H. H. Aung, and B. A. Myers, "Design requirements for more flexible structured editors from a study of programmers' text editing," in *CHI'05 extended abstracts on human factors in computing systems*, 2005, pp. 1557–1560.

[61] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 492–501.

[62] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: a review of the literature from an educational perspective," *Computer Science Education*, vol. 18, no. 2, pp. 67–92, 2008.