HookLens: Visual Analytics for Understanding React Hooks Structures

Suyeon Hwang* [1,2] Minkyu Kweon† [1] Jeongmin Rhee† [1] Soohyun Lee† [1]
Seokhyeon Park† [1] Seokweon Jung† [1] Hyeon Jeon† [1] Jinwook Seo‡ [1]
[1]Seoul National University [2]Samsung Electronics

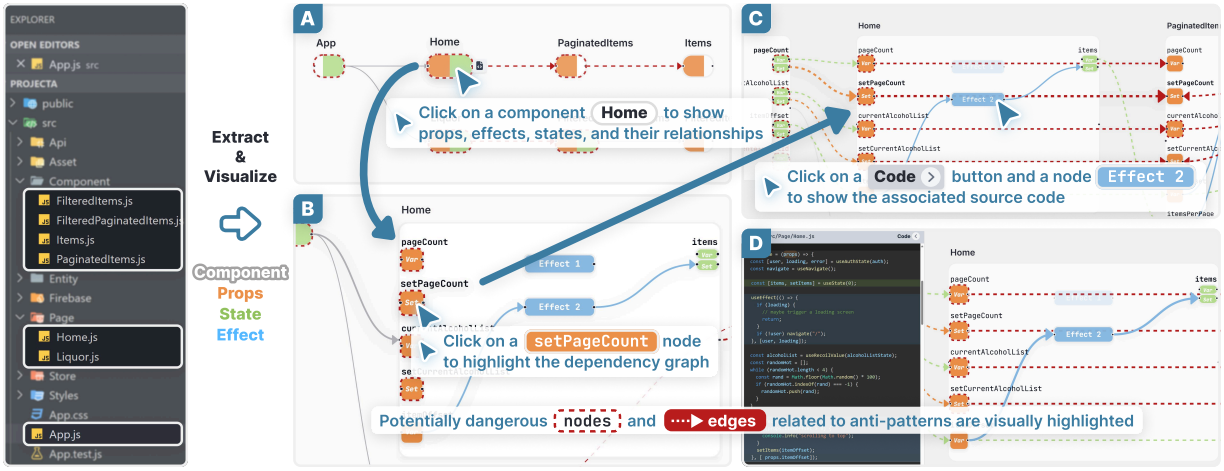

Figure 1: The overview of HookLens. The nested directed node-link diagram visualizes the structure of a *React* application. Nodes represent `Components`, `Props`, and `Hooks`—key elements for understanding *React*—and edges indicate their relationships. Nodes for `Props` and `Hooks` are nested within `Component` nodes and are initially hidden. Users can click a `Component` node to reveal its internal `Props`, `Hooks` and their relationships, and then examine these details (A). Users can also click a `Prop` or `Hook` node to highlight its relevant nodes and edges (B). Potentially problematic `Components`, `Props`, `Hooks`, and relationships are highlighted with red outlines and edges (C). Moreover, users can view the associated source code in the code viewer (D).

Abstract

Maintaining and refactoring *React* web applications is challenging, as *React* code often becomes complex due to its core API called `Hooks`. For example, `Hooks` often lead developers to create complex dependencies among components, making code behavior unpredictable and reducing maintainability, i.e., anti-patterns. To address this challenge, we present HookLens, an interactive visual analytics system that helps developers understand how `Hooks` define dependencies and data flows between components. Informed by an iterative design process with experienced *React* developers, HookLens supports users to efficiently understand the structure and dependencies between components and to identify anti-patterns. A quantitative user study with 12 *React* developers demonstrates that HookLens significantly improves participants' accuracy in detecting anti-patterns compared to conventional code editors. Moreover, a comparative study with state-of-the-art LLM-based coding assistants confirms that these improvements even surpass the capabilities of such coding assistants on the same task.

Index Terms: Visual analytics, Software visualization, Code analysis, Dependency graph visualization, React, Anti-pattern detection.

1 Introduction

React[1] is widely used in modern web development, but maintaining and refactoring *React* applications remains challenging. Unlike conventional web development, where the code that defines the structure, functionality, and design of user interface (UI) components is scattered across multiple files, *React* enables developers to implement these components as self-contained functions. By doing so, *React* helps developers build applications more intuitively and efficiently. Furthermore, `Hooks`, the core API of *React*, extend the functionality of each component, enabling the development of more dynamic and interactive applications. However, `Hooks` often introduce complex dependencies among components, creating anti-patterns that make code unpredictable, difficult to understand, and challenging to maintain [14, 12]. For example, when `Hooks` are used across hierarchically nested components, developers frequently pass data through intermediate components that do not directly use it; this anti-pattern forces them to review multiple components when modifying a single one.

To address this challenge, we propose HookLens, an interactive visual analytics system that supports developers in understanding the structure and dependencies of *React* code from the perspective of `Hooks`. We first conduct preliminary interviews with three experienced *React* developers to identify the core tasks involved in understanding and refactoring *React* applications. We then conduct an iterative design process with eight additional experienced *React* developers, enhancing the usability of HookLens and its effectiveness in visually exploring code structure and identifying anti-patterns. Our final prototype of HookLens enables users to analyze the structure of applications, from component hierarchies to code-level details, through details-on-demand interactions.

*e-mail: stom.hwang@{hcil.snu.ac.kr, samsung.com}
†e-mail: {mk, jmrhee, shlee, shpark, swjung, hj}@hcil.snu.ac.kr
‡e-mail: jseo@snu.ac.kr, corresponding author

[1] https://react.dev/

To demonstrate the effectiveness and usability of HookLens, we conduct a quantitative user study with 12 *React* developers of varying levels of expertise. Participants are asked to identify anti-patterns associated with `Hooks` in *React* code using both HookLens and a conventional code editor (*VS Code*). We further compare the result obtained from participants using HookLens with those produced by state-of-the-art LLM-based coding assistants, such as *Claude Code*, on the same task to examine the analytical capabilities of these assistants. The results indicate that HookLens substantially improves the accuracy in identifying anti-patterns compared to *VS Code*, and even outperforms LLM-based assistants, underscoring the continued importance of visual analytics in understanding complex *React* applications.

In summary, our contributions are as follows:

- We identify core tasks and design requirements to understand *React* applications based on interviews and an iterative design process.
- We introduce HookLens, an interactive visual analytics system that helps developers better understand the code structure of *React* applications and more effectively detect anti-patterns.
- We demonstrate the effectiveness and usability of HookLens through a user study with real-world *React* projects and a comparative study with LLM-based coding assistants.

HookLens is available at https://hook-lens.github.io/hook-lens/, where users can freely upload their codebase or connect their GitHub repository.

2 Background

We discuss the preliminaries essential for understanding the remainder of this paper.

2.1 React

React is one of the most widely used libraries for developing web applications [57]. *React* creates and manages individual UI elements as functions called `Components`. The way each `Component` manages its data and functionalities is defined by a core API in *React* called `Hooks`.

Components. `Components` are functions that define UI elements [13, 35]. `Components` can declare child `Components` and pass data to them via arguments called `Props`. By tracing these hierarchies of `Components`, developers can easily understand the application's UI and behavior.

Hooks. `Hooks` allow *React* applications to perform complex operations by defining how they manage information and handle external interactions, such as data fetching or user input (e.g., mouse clicks) [33]. Among various built-in `Hooks`, we focus on two fundamental `Hooks`: `State` and `Effect`. These two `Hooks` are introduced early in the official *React* tutorials [38, 39] and are essential for understanding both the life cycle of `Components` [36] and their rendering process [37]. In fact, our analysis of *Stack Overflow*[2] posts from 2020 to 2024 reveals that questions about these two `Hooks` account for 82% of all `Hooks`-related posts, indicating that developers frequently struggle to use them (Figure 2). The two `Hooks` are described as follows:

`State` Hook, which can be defined using a function named `useState`, manages data within a `Component`. The value stored in a `State` Hook can be accessed within the corresponding `Component` and passed to its child `Components` through `Props`. When this value is modified by external sources, such as user interactions, the `Hook` triggers a re-render of the `Component` and its child `Components` to reflect the update. In this way, the `State` Hook enables developers to build dynamic and interactive applications that respond to data changes.

[2] https://stackoverflow.com/

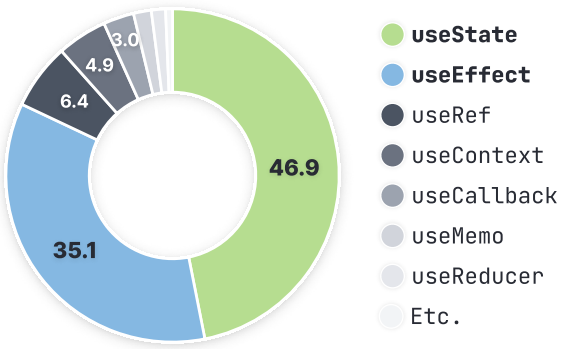

Figure 2: Ratio of posts about each built-in `Hook` on *Stack Overflow* (2020–2024). Data are obtained by querying posts tagged with *reactjs* that mention the name of each `Hook` using the *Stack Exchange API* (https://api.stackexchange.com/). `useState` and `useEffect` denote the functions for the `State` and `Effect` Hooks, respectively.

`Effect` Hook, provided by the `useEffect` function, defines how a `Component` responds to changes in external sources. This `Hook` allows developers to specify the operations to perform (logic) and the conditions that trigger them (dependencies), which typically depend on the values of `State` Hooks or `Props`. For example, when a value of `State` Hook included in the dependencies of an `Effect` Hook changes, the associated logic is executed. The `Effect` Hook thus enables *React* applications to perform operations beyond re-rendering, such as synchronizing data with other `Components` or external sources.

We refer to `State` Hook and `Effect` Hook simply as `State` and `Effect` hereafter.

2.2 Anti-Patterns in React

Although `Hooks` are powerful for developing interactive and dynamic applications, unmanaged or excessive use of them often leads to undesirable complexities and problems, known as anti-patterns [14, 12]. Various anti-patterns are identified in *React* development [12, 43, 14, 49], but we mainly focus on three major anti-patterns related to `States` and `Effects` [34, 40]. The following are descriptions of these three anti-patterns (Figure 3):

Unreferenced States and Props. This anti-pattern denotes a code pattern in which `State` values or `Props` are either unused within their defining `Components` or merely passed to another `Component` without being utilized. This pattern can increase memory consumption and introduce unnecessary code, potentially leading to additional anti-patterns.

Prop drilling. This pattern occurs when `State` values are passed through multiple `Components` without being directly used within them. It introduces unnecessary dependencies and data flows among `Components`, making the code more difficult to maintain and the flow of data harder to trace.

Effect modifying parent States. This pattern refers to code in which the `State` values of a parent `Component` are modified by the execution of an `Effect` defined in one of its child `Components`. Such a pattern complicates the understanding of the application's behavior due to unpredictable `State` values changes, often leading to unintended outcomes or bugs.

3 Related Work

We discuss prior studies related to HookLens. We first review research on analyzing *React* applications from various perspectives, followed by studies on software visualization that aim to analyze and understand code.

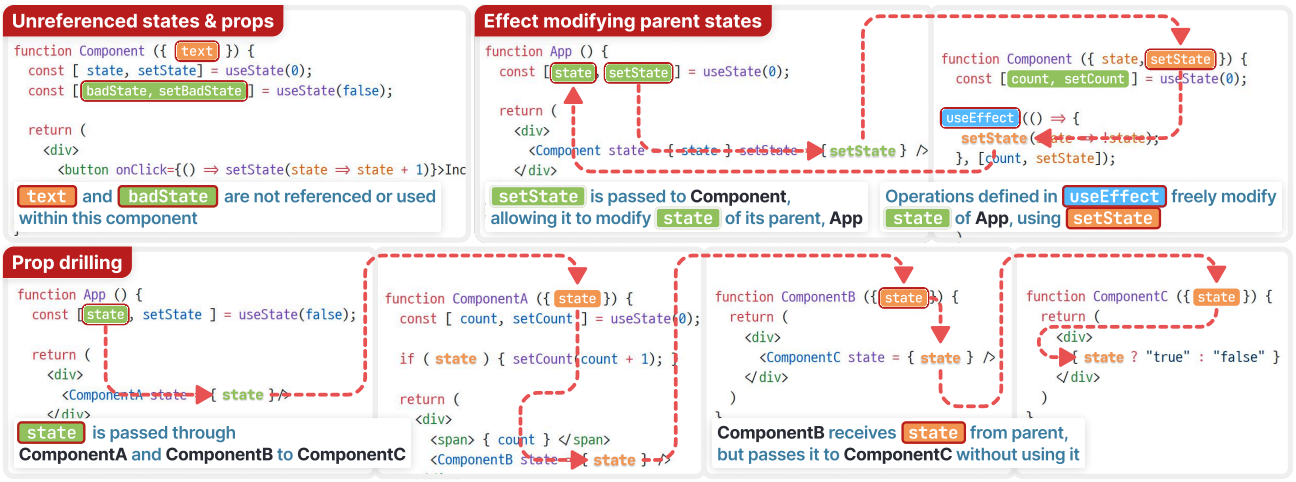

Figure 3: Examples of three anti-patterns. Each code snippet illustrates *Unreferenced States and Props*, *Effect modifying parent States*, and *Prop drilling*, respectively. In each example, relevant `States`, `Effects` and `Props` within `Components` are highlighted with red outlines, while red edges across `Components` indicate `State` propagation paths.

3.1 React Application Analysis

React has become widely used for web development, attracting research attention. For example, Sharma et al. [53] compare the performance of *React* with other frameworks such as *Angular*, while *ReactAppScan* [18] introduces a tool for detecting security vulnerabilities. Despite these efforts, our understanding of addressing the complexity arising from the architecture of `Components` and `Hooks` remains limited, particularly in supporting maintenance tasks. To the best of our knowledge, *React-tRace* [28] is the only attempt to address this gap. It analyzes the semantics of *React* code, focusing on the rendering process to support UI debugging. However, interpretability remains an issue, as users cannot readily locate the specific code where the problems originate. Similarly, though studies analyze *React* code to identify and mitigate anti-patterns [14, 13, 44], detecting anti-patterns across multiple files—particularly those involving `Hooks`—remains challenging. Visual analytics approaches, such as *React Bratus* [5] and Tarner et al. [59], aim to address this limitation by visualizing hierarchical `Component` structures, but developers still struggle to understand the behavior of `Hooks` and the dynamic nature of *React* applications.

Our contribution. We propose a visual analytics system that assists developers in understanding code structures and detecting anti-patterns in *React* applications. Unlike prior studies [5, 59] that primarily focus on `Component` structures or execution outcomes, our approach centers on the interactions among `Components` and `Hooks`, similar in spirit to *React-tRace* [28]. However, rather than analyzing rendering semantics, we focus on uncovering anti-patterns that emerge within *React* environments shaped by `Hooks`. Through this approach, developers can gain a deeper understanding of how `Hooks` influence the behavior of *React* applications and more effectively explore anti-patterns related to `Hooks`.

3.2 Software Visualization

The utility of software visualization. Visualization and visual analytics are widely used to analyze, debug, and maintain software [7, 30, 25, 24]. Prior studies demonstrate that software visualization helps developers understand concepts that bind to certain environments or programming languages. For example, *RustViz* [1] visually illustrates difficult-to-understand concepts in *Rust*, such as *ownership* and *borrowing*. *InterLink* [29] reveals the relationship between code and text cells in computational notebooks (e.g., Jupyter). Software visualization can also uncover hidden information that is hard to perceive in raw source code, such as module dependencies or data flows. For instance, Gouveia et al. [17] represent the execution counts for each code segment using visual diagrams to support runtime debugging. *GraphBuddy* [6] and *DeJEE* [54] visualize the dependencies between modules and libraries in *Java* environments. Similarly, *Reactive Inspector* [51] and *RxFiddle* [4] provide visual debugging tools that visualize signal flows and dependencies in reactive programming environments.

The utility of node-link diagrams in software visualization. While prior studies employ diverse visual idioms for software visualization, node-link diagrams stand out as a particularly effective and widely adopted approach. Many of the aforementioned systems either directly employ node-link diagrams [6, 54, 51, 4] or incorporate them into their visual designs [1, 29]. This is because node-link diagrams intuitively reveal relationships such as dependencies and data or control flows among program elements (e.g., functions, classes, and modules) [50]. For instance, *E-Quality* [59] visualizes code quality metrics on node-link diagrams by following relationships among classes to support refactoring and assess maintainability. In contrast, *REACHER* [27] and Ishio et al. [21] employ node-link diagrams to assist program comprehension by revealing control and data flows within the source code. Beyond traditional programming environments, *Rapsai* [9] and *Misty* [31] extend this approach to higher-level development tasks such as constructing machine learning pipelines and prototyping mobile UIs. More recently, node-link diagrams are employed to support LLM-based code generation, visualizing user prompts instead of source code to help users interpret and guide the code generation process [10, 69, 47].

Our contribution. Building on the proven utility of visualization in software analysis and maintenance, we employ software visualization—particularly node-link diagrams—to support code comprehension and anti-pattern detection based on `Components` and `Hooks`. As prior works demonstrate the effectiveness of node-link diagrams across diverse software analysis, development and maintenance tasks, we extend conventional node-link diagrams into a nested and interactive structure. Our approach enables developers to more effectively explore the hierarchical structure of `Components` and uncover hidden dependencies introduced by `Hooks` within these hierarchies.

4 Tasks and Design Requirements

We introduce the tasks and design requirements of HookLens.

4.1 Tasks

We identify two core tasks through a semi-structured preliminary interview. We recruit three *React* developers (three males, aged 28, 30, and 34) with over two years of experience and intermediate proficiency. All participants have experience working on at least two projects, either individually or in small teams of up to four members. We ask participants about their use of `Hooks`—particularly `State` and `Effect`—in maintaining and refactoring code. Each interview lasts up to 35 minutes.

(T1) Managing States and their propagation among Components. `States` within each `Component` are closely tied to its execution logic and rendering outcomes, and child `Components` may also depend on certain `States` from their parent `Components` for their own execution. While `States` serve as essential and powerful tools for creating dynamic and interactive `Components`, developers often fail to manage them properly or tend to overuse them. In particular, developers often propagate `State` values and their setter functions redundantly across multiple child `Components` for convenience (i.e., *prop drilling*) Such unmanaged and widely propagated `States` make it difficult to trace their origins, and as the number of `Components` increases, even minor code changes may require reviewing or modifying a large number of `Components`, thereby reducing maintainability. Therefore, developers need to understand where `States` are defined, how they are propagated, and which `Components` are affected by their changes.

(T2) Tracking the chains of Effects. As the logic defined in `Effects` can execute independently from the rendering process of *React*, developers need to track and understand how these `Effects` are executed. However, when a `Component` handles too many functionalities or becomes overly complex, excessive use of `Effects` may occur, forming long chains of `Effects`. In some cases, these `Effects` even propagate upward through the `Component` hierarchy, affecting parent `Components` (i.e., *Effect modifying parent states*). Since these `Effects` depend on runtime conditions and are triggered by external sources across multiple `Components`, predicting their outcomes is challenging. Such unpredictability can introduce unintended behaviors that are difficult to trace and fix, and can further degrade application performance. Therefore, developers should carefully track and understand the behaviors and outcomes defined by these `Effects`.

4.2 Design Requirements

We formulate design requirements to support developers in effectively performing the core tasks identified in Section 4.1.

(DR1) Focus on Props, States, and Effects within Components. Although diverse approaches exist for analyzing *React* applications [5, 59], developers primarily reason about `Component` behavior through `Props`, `States`, and `Effects` (T1-–T2). This is because `States` and `Effects` play crucial roles in determining application behavior (e.g., rendering process and `Component` life cycle), while `Props` serve as essential links to trace data and control flow across `Components`. Therefore, HOOKLENS should focus on these three elements to help developers understand data flow, debug logic, and identify dependencies.

(DR2) Deliver the hierarchical structure of Components. *React* applications operate and render based on the hierarchical structures of `Components`, where `State` values propagate along these hierarchies through `Props`. To understand such operations and propagation (T1), developers need to examine the hierarchical relationships among `Components`. However, these hierarchies are often deeply nested and difficult to infer from source code alone. Therefore, HOOKLENS visually represents these hierarchical structures to help developers comprehend how data and control flow through the `Component` hierarchy.

(DR3) Explain the dynamic interactions among Props, States, and Effects. Beyond identifying these core elements (DR1), understanding how they dynamically interact is key to reasoning about *React* application behavior. In *React* applications, data and control flows emerge from the interactions among `Props`, `States`, and `Effects` (Section 2.1). For example, updates to `States` of the parent `Component` may propagate as `Props` to child `Components` (T1), triggering `Effects` that perform additional operations (T2). Therefore, HOOKLENS should visualize these dynamic dependencies to help developers understand how these elements influence the overall behavior of applications.

(DR4) Reveal potential anti-patterns. While anti-pattern detection is crucial for refactoring [23, 15], it remains challenging in *React* code, especially as applications scale in size and complexity. In particular, anti-patterns related to `Hooks` often span multiple `Components` and files (e.g., *prop drilling* and *Effect modifying parent states*), making them difficult to detect and trace. These complexities hinder issue resolution and reduce code maintainability. Therefore, HOOKLENS should highlight problematic patterns in the use of `Hooks` to help developers identify and address potential issues and refactoring opportunities.

5 Iterative Design Process

We adopt an iterative design process to develop HOOKLENS.

Designing the initial prototype. We first build an initial prototype based on the design requirements (Section 4.2) to explore potential improvements in visual representations and interactions. Our prototype visualizes the hierarchical structure of `Components` (DR2) and the relationships among `States` and `Effects` (DR1, DR3) through a node-link diagram. Moreover, the prototype can automatically detect and visually highlight *unreferenced states and props* anti-patterns (DR4).

Receiving feedback from experienced *React* developers. We conduct feedback sessions with eight experienced *React* developers (eight males, aged 22–31 [27$\pm$5]), each with more than two years of experience. During each session, we explain the goal of HOOKLENS to the participants and demonstrate its functionality using a sample project that contains several anti-patterns, such as *unreferenced states and props* and *prop drilling*. We then present a simple scenario in which participants trace `States` within a specific `Component` and identify the dependencies of an `Effect`. Throughout the session, participants are encouraged to freely share their opinions and suggestions regarding the prototype.

Designing the final prototype. Based on the feedback, we refine the final prototype of HOOKLENS with two major improvements.

(I1) Highlight areas of interest. While most participants agree that the prototype effectively reveals the structural relationships among `Components`, `States`, and `Effects`, they report difficulties in keeping track of which elements they are focusing on during exploration. This issue becomes particularly critical in larger projects, where the large number of nodes and edges makes navigation more challenging. To address this, we enable users to interactively select and visually emphasize areas of interest.

(I2) Integrate a code viewer for detailed analysis. Participants emphasize the need for an integrated code viewer to support detailed analysis of the application. Although they can identify `Components` or `Hooks` where issues exist or validation is required, the prototype alone makes it difficult to review exact execution timings or data changes. In particular, some participants suggest adding a feature that directly displays the code related to the focused `Components` or `Hooks`. We therefore integrate a code viewer that highlights the code corresponding to the elements currently focused in the visualization.

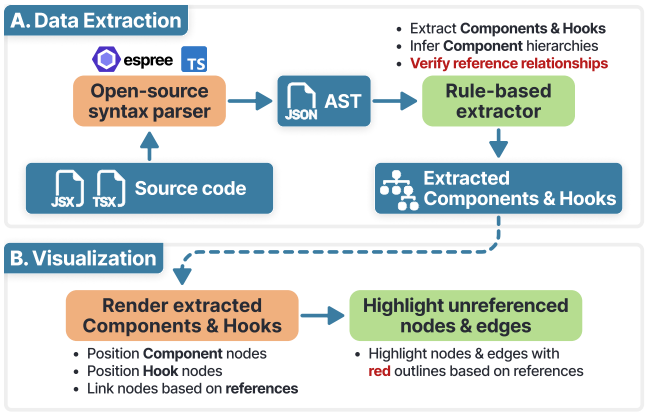

Figure 4: Data extraction and visualization pipeline of HOOKLENS. HOOKLENS parses source files into abstract syntax trees (ASTs) using *Espree* and the *TypeScript* compiler API, then extracts `Components`, `Hooks`, and their reference relationships with predefined rules. It visualizes the extracted data as a node-link diagram, highlighting invalid references in red to facilitate anti-pattern detection.

6 HOOKLENS: AN INTERACTIVE VISUAL ANALYTICS SYSTEM

We present HOOKLENS, an interactive visual analytics system that helps developers understand the overall structure of *React* applications through `Components` and `Hooks`, and identify related anti-patterns. The system extracts key elements such as `Components`, `Props`, and `Hooks` from a given *React* project and visualizes their relationships through an interactive node-link diagram.

6.1 Data Extraction Pipeline

Given a *React* project, HOOKLENS extracts `Components`, `Props`, and `Hooks` (i.e., `States` and `Effects`) and builds structured data based on the relationships among these elements (DR1; Figure 4A). To ensure fast and consistent extraction of these elements, we develop a rule-based extractor module. We first leverage (1) the compiler API of *TypeScript* [41] and (2) the open-source *JavaScript* parser, *Espree* [11], to convert the source code into abstract syntax trees (ASTs). We adopt ASTs because they are widely used for static code analysis [64, 62] and facilitate systematic exploration of source code. Building on these ASTs, we implement extraction logic that detects `Components`, `Props`, `States`, and `Effects`, and constructs their relationships. This logic follows predefined rules to identify target elements and trace their dependencies. Furthermore, the extractor verifies reference relationships among the extracted elements to identify unreferenced or unused `Components`, `Props`, and `States`, as well as `Effects` that introduce backward dependencies to their parent `Components`. As a result, the extractor module builds a graph-structured dataset that represents the extracted elements and their relationships.

6.2 Visual Analytics Design

We describe how HOOKLENS visualizes code structures and anti-patterns related to `Hooks` (Figure 4B), and how it supports interactive exploration (Figure 1).

Visualization for Components and Hooks. HOOKLENS visualizes the extracted `Components`, `Props`, and `Hooks`, along with their relationships, using an interactive node-link diagram. Since node-link diagrams align with how developers conceptualize function calls and data flows [50], HOOKLENS adopts this approach to facilitate users' understanding of `State` value propagation (T1) and chains of `Effects` (T2). Nodes in the diagram represent extracted elements—`Components`, `Props`, `States`, and `Effects` (DR1)—while edges indicate relationships such as dependencies of `Effects` and data flows between `States` and `Props` (DR3). In particular, `State`, `Prop`, and `Effect` nodes are placed within their corresponding `Component` node to indicate that these elements are defined by the `Component`. To clearly distinguish different types of nodes and edges, we use distinct hues with consistent saturation levels, thereby avoiding unintended implications of ordinal attributes [61, 32]. Furthermore, HOOKLENS highlights potential anti-patterns (Figure 5) using animated red outlines and edges (DR4; Figure 1B). Based on the reference relationships verified during the extraction process (Figure 4), the system emphasizes unreferenced or unused elements as well as `Effects` that introduce backward dependencies to their parent `Components`. These visual cues make potential anti-patterns visually salient, as all target anti-patterns in this study arise from reference relationships and their associated data or control flows. We adopt red as it is a widely recognized visual cue for warnings in development environments.

Interactive exploration. To help users explore relationships among `Hooks` without cognitive overload, we design HOOKLENS following Shneiderman's visual information seeking mantra [55]. The system first presents an overview of `Components` and their relationships, providing a high-level understanding of the project. `Component` nodes are arranged horizontally by hierarchy, enabling developers to view deeply nested structures within a single screen (DR2). Users can then reveal `States`, `Effects`, `Props`, and their relationships on demand. Clicking on a `Component` node expands it to show its internal `States`, `Effects`, and `Props` (Figure 1A and B). When related `Components` are also expanded, the system visualizes edges representing the propagation of `State` values and `Props` (DR3). Clicking a `State`, `Effect`, or `Prop` node highlights its related data and control flows while dimming others (I1; Figure 1C). Users can further navigate the diagram using pan and zoom interactions. For deeper inspection, HOOKLENS integrates a code viewer that automatically locates and highlights the source code corresponding to the selected node (I2; Figure 1D).

7 USER STUDY

We conduct a user study to evaluate the effectiveness and usability of HOOKLENS in performing the target tasks (Section 4.1), comparing it with a popular code editor (*VS Code*).

7.1 Objectives and Design

We aim to evaluate how effectively HOOKLENS supports developers in performing the target tasks (Section 4.1) and how usable the system is for understanding and maintaining *React* applications. To this end, we recruit *React* developers and measure their performance in terms of task accuracy. We further assess the usability of the system through the *System Usability Scale* (SUS) and post-study interviews.

Participants. We recruit 12 *React* developers (eight males and four females, aged 21–30 [25±9]) through snowball sampling. In particular, we recruit participants with diverse levels of expertise in *React* development to examine whether HOOKLENS effectively supports both novice and intermediate developers. Six participants (P1–P6) have less than two years of experience, while the remaining six (P7–P12) have more than two years of experience, consistent with the conditions of the preliminary interviews and feedback sessions. All participants are compensated with the equivalent of USD 10.

Tasks. We ask participants to detect anti-patterns described in Section 2.2 within a *React* project. Since detecting anti-patterns is a common task in code maintenance and refactoring [23, 15], we adopt it as the basis of our study. Participants are asked to identify as many anti-patterns related to the use of `States` and `Effects` as possible within a 10-minute time limit. Before each session, we inform participants about the three target anti-patterns described in

Table 1: Comparison of two open-source projects used in the study. We select the projects that are similar in terms of their size and the number of existing anti-patterns.

Metric	*Confides*	*paper_vis*
Size		
Number of JSX files	29	30
Number of components	25	33
Total lines of code (JSX)	2707	3937
Anti-patterns		
Unreferenced states & props	41	32
Prop drilling	11	11
Effect modifying parent states	2	2

Section 2.2 and instruct them to detect only these patterns. Participants verbally report the presence of each anti-pattern and specify the associated `Components`, `Props`, `States`, and `Effects`. We measure precision and recall in detecting anti-patterns, using a ground truth established through careful code examination. To ensure a fair comparison and focus on the core capabilities of HOOKLENS, we consider only reports involving `States` and `Effects` when computing these metrics. To ensure objectivity, the ground truth is constructed based on the formal definitions and code-level criteria of each anti-pattern (Section 2.2) and validated through independent review and consensus between two authors. Reports related to other variables (e.g., variables from `useRef` or general *JavaScript* variables) are excluded from the evaluation, as participants are instructed in advance to focus on the target patterns.

Baseline. To evaluate the cognitive advantages of visual analytics over text-based exploration, we select *VS Code* as the baseline. In addition to being one of the most widely used development tools [57], all preliminary interviewees and 11 out of 12 study participants report using *VS Code* together with extensions such as *eslint-plugin-react-hooks*[3]. However, *eslint-plugin-react-hooks* primarily enforces basic implementation rules——such as ensuring consistent `Hooks` call order——rather than detecting complex anti-patterns that span multiple components. As a result, identifying such anti-patterns in text-based environments often requires extensive manual tracing, which imposes a significant cognitive burden. Therefore, to compare HOOKLENS against this manual process, we provide a baseline environment with only the default *IntelliSense* features enabled, allowing participants to use standard navigation functions (e.g., *Go to Definition* and *Find All References*) to manually trace dependencies.

Sample projects. We select two open-source projects (*Confides* [65] and *paper_vis* [63]) to simulate real-world tasks. Many popular open-source *React* projects are developed by large teams and undergo extensive maintenance, during which such anti-patterns are often controlled or resolved, making them less suitable for our study tasks within a limited time budget. Therefore, we focus on projects developed by individuals or small teams, which are academic prototypes. Such projects are typically developed as proof-of-concept systems and thus tend to contain a sufficient number of representative anti-patterns, making them well suited for controlled evaluation of anti-pattern detection. To mitigate learning effects that could bias tool performance, we employ two separate projects with comparable task complexity, each containing a similar number of files, `Components`, and anti-patterns (Table 1).

Procedure. After participants consent to their participation, we first explain HOOKLENS and its key features (e.g., visualization and available interactions). We then provide participants approximately 20 minutes to explore and practice with the system, where they are allowed to freely ask questions. After the practice session, we spend about 5 minutes introducing the study tasks and the three target anti-patterns (Section 2.2), presenting examples of each in both code (Figure 3) and visualization (Figure 5) within HOOKLENS. Subsequently, participants perform the task in two conditions: using *VS Code* and HOOKLENS, with each condition lasting 10 minutes (20 minutes in total). To control for ordering effects, half of the participants begin with HOOKLENS, while the others start with *VS Code*. We also assign different projects for each condition to control learning effects. After completing both sessions, participants fill out the SUS questionnaire and participate in a post-study interview for approximately 15 minutes to provide qualitative feedback on system usability and overall satisfaction. In total, the entire study session takes approximately one hour.

[3] https://react.dev/reference/eslint-plugin-react-hooks

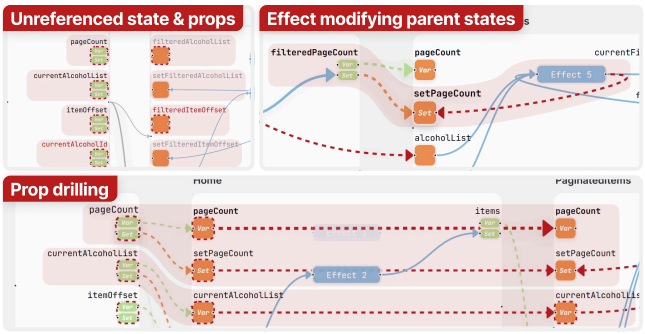

Figure 5: Types of anti-patterns revealed by HOOKLENS. Anti-patterns are highlighted with translucent red backgrounds: *Unreferenced states and props*, shown as red outlined `State` and `Prop` nodes; *prop drilling*, represented by red edges between `Prop` nodes; and *effect modifying parent states*, represented by red edges originating from `Effect` nodes.

7.2 Quantitative Results

Our study results confirm the effectiveness of HOOKLENS in supporting participants in detecting anti-patterns. They further imply the superiority of HOOKLENS in user experience compared to a conventional code editor.

Accuracy in detecting anti-patterns. We calculate F1-scores for anti-pattern detection based on participants' responses in tasks performed with both *VS Code* and HOOKLENS. A Mann–Whitney U test on these scores shows that participants achieve significantly higher task accuracy with HOOKLENS than with *VS Code* (Table 2). Further statistical analysis by expertise level suggests that HOOKLENS effectively supports both novice and intermediate developers in identifying anti-patterns, although differences between the two groups are also observed (Figure 6). We further discuss these differences between novice and intermediate developers in the qualitative findings section (Section 7.3).

Usability of HOOKLENS. The SUS results indicate that HOOKLENS shows high usability for the given tasks, with an average score of 76.7. This score exceeds the standard benchmark of 68, suggesting that participants generally found the system easy to learn and use [52]. These findings are also consistent with the qualitative findings obtained from the post-study interviews (Section 7.3).

7.3 Qualitative Findings

We present the qualitative findings derived from task observations and post-study interviews. These findings provide plausible explanations for the quantitative outcomes and further highlight both the strengths and potential areas for improvement of HOOKLENS.

Table 2: Performance comparison between HOOKLENS and *VS Code* in detecting different anti-patterns. Values are reported as mean ± standard deviation. All results are statistically significant ($p \ll 0.01$). We color the table cells in red with an opacity scale where lower opacity represents lower accuracy (linear gradient between 0 and 1).

Anti-pattern	HOOKLENS			*VS Code*		
	Precision	Recall	F1	Precision	Recall	F1
Unreferenced states & props	.968 ± .098	.509 ± .311	.614 ± .272	.669 ± .322	.147 ± .111	.219 ± .153
Prop drilling	.938 ± .121	.568 ± .255	.669 ± .213	.492 ± .457	.197 ± .243	.264 ± .294
Effect modifying parent states	.889 ± .283	.792 ± .320	.817 ± .284	.181 ± .369	.167 ± .312	.160 ± .316

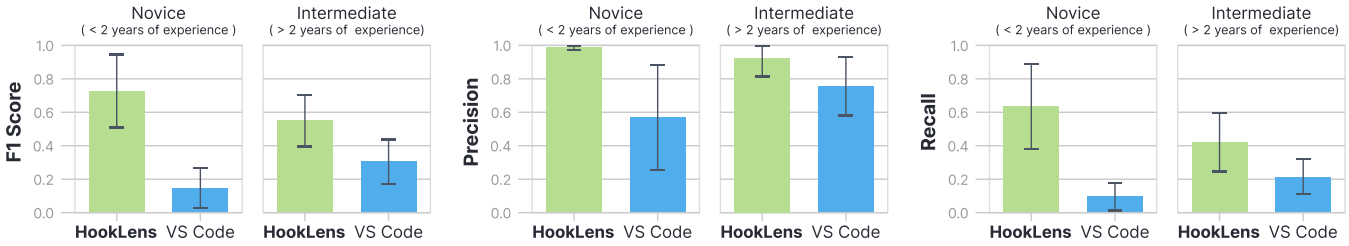

Figure 6: Accuracy comparison between HOOKLENS and *VS Code* in detecting anti-patterns, separated by proficiency. Novice participants have less than two years of experience, while intermediate participants have more than two years. Using HOOKLENS resulted in statistically significant improvements in F1-score and recall for both novice ($p \ll 0.01$) and intermediate ($p \ll 0.05$) groups, though not in precision.

Advantages of visual analytics in understanding React applications. Most participants agree that the visualization provided by HOOKLENS is highly effective for understanding the overall structure and the relationships among `Components` and `Hooks` across multiple files, which are often difficult to grasp with conventional code editors. For example, several participants (e.g., P2, P9, P10) mention that HOOKLENS helps them quickly gain an overview of unfamiliar projects, as its visualization supports comprehension without requiring line-by-line code inspection. P2 adds that the spatial positioning of `Component` nodes intuitively conveys structural hierarchy, making it easier to interpret the layout. P9 notes that this advantage could substantially support refactoring legacy code within their organization. Several participants (e.g., P3, P6, P8, P12) also highlight that HOOKLENS facilitates identifying specific files and `Components` relevant to their tasks, even when multiple `Components` are defined in a single file or when directory structures are complex. Participants further report that HOOKLENS is effective in tracing `State` propagation and `Effect` dependencies across `Components`. P7 notes that it helps investigate how `Components` and `States` reference each other and which `Components` are designed for shared use across `Components`. Similarly, P3 and P4 mention its usefulness for managing `States`, such as locating where they are defined and which `Props` deliver their values, while P8 finds that tracing `Effects` is straightforward by simply following the directed edges. Additionally, P9 and P12 emphasize that HOOKLENS makes it much easier to trace relevant `Components`, `States`, and `Effects` compared with conventional editors, which require developers to manually search across multiple files.

Usability in detecting anti-pattern. Both novice (e.g., P1, P3, P6) and intermediate (e.g., P8, P11, P12) participants find that HOOKLENS is useful for identifying anti-patterns. They note that the red highlights in HOOKLENS effectively draw their attention, enabling them to quickly recognize the relevant `Components`. In particular, P6 and P11 emphasize that analyzing the relationships among `Components` and `Hooks` should take precedence over detailed source code analysis when detecting such anti-patterns, and that HOOKLENS is therefore especially helpful by filtering out unrelated `Components`. Meanwhile, P3 notes that when he encounters *prop drilling*, HOOKLENS facilitates exploration by visualizing the detailed references of `Props` and helping prioritize which ones may pose potential risks.

Differences in use between novice and intermediate participants. As shown in Figure 6, novice participants perform better with HOOKLENS than intermediate participants, and the observed p-values indicate stronger statistical significance for the novice participants. During the study tasks with HOOKLENS, most novice participants rely heavily on the visualizations provided by HOOKLENS. For example, P1 relies entirely on visual exploration and the visual cues provided by HOOKLENS to report answers, without directly inspecting the source code. Similarly, most other novice participants (e.g., P2, P4–P6) primarily rely on visual exploration to complete the tasks rather than inspecting the source code, which leads to faster anti-pattern detection. Exceptionally, P3 tends to inspect the source code more frequently than other novice participants, but consistently achieves low accuracy in both the baseline and HOOKLENS conditions. Meanwhile, intermediate participants tend to use the visualization as a reference while repeatedly reviewing the source code throughout the tasks. They generally achieve higher accuracy than novice participants in the baseline condition, suggesting that they are more familiar with navigating source code. This greater familiarity is further reflected in their review of the source code even when performing tasks with HOOKLENS, which is associated with longer anti-pattern detection times compared to novice participants. In fact, we observe that P8 and P9 keep the source-code view open throughout most of the task execution.

Notably, most intermediate participants (e.g., P8, P10–P12) note that, unlike *VS Code*, HOOKLENS provides limited features for direct code inspection and detailed behavior verification, making it challenging to fully grasp application logic. By contrast, novice participants (e.g., P1–P3) report that they can understand application behavior more easily, likely because they usually work on projects where the interactions among `Components`, `States`, and `Effects` are relatively simple.

Limitations in supporting real-world development workflows. Although HOOKLENS facilitates users in performing core tasks, participants also point out several limitations. First, some participants note that although HOOKLENS is effective for examining code structures and identifying anti-patterns, it provides limited support for analyzing runtime behavior. P8 emphasizes that detailed analysis becomes feasible only after gaining sufficient familiarity with the system, and P11 adds that HOOKLENS restricts source code exploration compared to *VS Code*. Several participants

also suggest that HOOKLENS should include additional features to better support real-world development. For example, P9 and P10 express concern that HOOKLENS does not support external state management systems such as *Redux*, which are widely used in practice and can help mitigate *prop drilling*. Finally, P12 points out that tracing relationships in regions with many intersecting edges can be overwhelming, highlighting the need to improve visual scalability to better support larger projects.

8 COMPARATIVE EVALUATION WITH LLM-BASED CODING ASSISTANTS

To further assess the effectiveness of HOOKLENS, we compare its performance against state-of-the-art LLM-based coding assistants. Specifically, we evaluate how well these assistants identify anti-patterns related to `Hooks` and whether their detection performance is comparable to that of human-in-the-loop visual analytics.

8.1 Objectives and Design

We evaluate whether HOOKLENS outperforms state-of-the-art LLM-based coding assistants in detecting anti-patterns. To this end, we select representative assistants and measure their task accuracy using the same procedure as our user study, comparing the results with those of the participants.

LLM-based coding assistants. We select three state-of-the-art LLM-based coding assistant tools to evaluate their capabilities in detecting anti-patterns. To ensure fairness and consistency, we focus on cloud-based commercial tools that demonstrate high performance on the de facto benchmark, *SWE-bench*[4] [22], which evaluates models on code understanding and software engineering problem solving. The selected assistants are as follows:

- ***Claude Code*** [2] provides the *claude-sonnet-4* (model: claude-sonnet-4-20250514, released May 2025) and *claude-opus-4.1* (model: claude-opus-4-1-20250805, released Aug 2025) models, which currently achieve the highest scores on *SWE-bench*.
- ***Codex CLI*** [45] offers one of the latest high-performing models, *GPT-5* (released Aug 2025).
- ***Gemini CLI*** [16] provides the *gemini-2.5-pro* (released Aug 2025) model, which performs slightly below the other two tools but still ranks highly on *SWE-bench*.

Procedure. We conduct the same tasks as in the user study (Section 7) for each coding assistant and compare the results with those of human participants using HOOKLENS. For each assistant, we perform six independent trials on the two projects used in the user study and count the number of detected anti-patterns using the same ground truth. To ensure equivalent conditions, we grant each assistant full access to all project files, reset its state before each trial, and provide an identical prompt (appendix A) that includes a code-level one-shot example of each anti-pattern, replicating the conditions given to user study participants.

8.2 Results

Compared to participants using HOOKLENS, most LLM-based assistants struggle to detect anti-patterns (Figure 7). Except for *Codex CLI* with *GPT-5*, all tools perform worse than even the *VS Code* baseline. Although *GPT-5* achieves relatively better performance than other LLMs, its precision remains statistically lower than that of HOOKLENS. These results are consistent with prior studies showing that LLMs often face difficulties in analyzing code with extensive context [67, 60, 56]. In particular, the anti-patterns examined in this study—spanning multiple files and requiring contextual reasoning—further expose this limitation. For example, we observe that all four LLMs frequently produce false positives by incorrectly identifying imported relationships that are never actually referenced (Figure 8A). We also observe that LLMs repeatedly fail to detect certain *prop drilling* patterns that are identified by most participants using HOOKLENS (Figure 8B). Beyond these examples, LLMs often report partially correct or incorrect results, such as identifying only sub paths of anti-patterns or inferring incorrect data flows. Despite the growing capabilities and popularity of such assistants, these findings underscore the continued need for human-in-the-loop visual analytics tools like HOOKLENS, as further discussed in Section 9.4.

[4] https://www.swebench.com/

9 DISCUSSION

We discuss the limitations of our study and directions for future work.

9.1 Scalability and Project Context

Our study evaluates HOOKLENS only on relatively small projects (25–33 components), which limits the generalization of the results to larger-scale projects. Although HOOKLENS employs a two-level nested node-link diagram, one participant reports visual clutter when analyzing the study projects (Section 7.3), suggesting scalability challenges as project size increases. Therefore, future work should investigate alternative visualization (e.g., treemap, circular tree, and icicle tree) [3] and interaction techniques to better manage visual complexity in larger projects [19]. In addition, our evaluation focuses exclusively on unfamiliar projects. Future studies should examine whether the observed effectiveness of HOOKLENS generalizes to projects that developers are already familiar with, which more closely reflect real-world development settings.

9.2 Supporting Real-World Workflows for Semantic Validation

Although our results suggest that HOOKLENS effectively supports understanding code structures and detecting anti-patterns that span multiple `Components`, several refactoring tasks remain challenging. These include detecting a broader range of anti-patterns, performing fine-grained code-level reasoning within individual `Components`, reasoning about runtime behavior (e.g., execution timing and data changes), and conducting semantic validation of detected anti-patterns. Semantic validation is particularly important, as some detected anti-patterns may not represent actual issues without considering runtime context or developer intent. In this regard, the primary contribution of HOOKLENS lies in its ability to visually organize analysis results, reducing the cognitive burden of manually tracing control and data flows across `Components`. However, our user study shows that most intermediate participants continue to rely on direct source code inspection when performing study tasks, indicating that source code analysis still plays an important role in their workflows and mental models (Section 7.3). Therefore, to better support real-world development workflows, HOOKLENS should be provided as an integrated extension or plugin to conventional code editing tools, enabling not only anti-pattern detection but also semantic validation and other intermediate-level refactoring tasks. Prior studies show that implementing visualization tools as extensions or plugins to existing development environments is an effective approach for integrating them into established workflows [29, 6, 3, 54, 21, 17].

9.3 Extending Support to the Broader React Ecosystem

While HOOKLENS focuses on the `State` and `Effect Hooks`, which are fundamental to understanding *React* applications, extending support to other built-in `Hooks` and external state management systems remains an important direction for future work. As indicated by participants in our user study (Section 7.3), many developers rely on other built-in `Hooks`, such as the `Context`

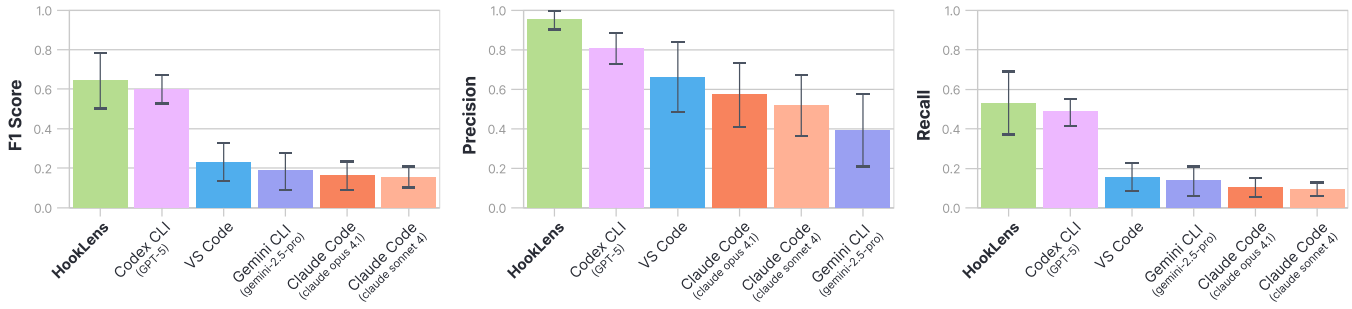

Figure 7: Accuracy comparison between human developers and LLM-based coding assistants in detecting anti-patterns. HOOKLENS shows substantially higher mean accuracy than other tools, which is statistically significant ($p \ll 0.01$) except in the case of *GPT-5*. In terms of precision, HOOKLENS also outperforms *GPT-5* ($p \ll 0.01$).

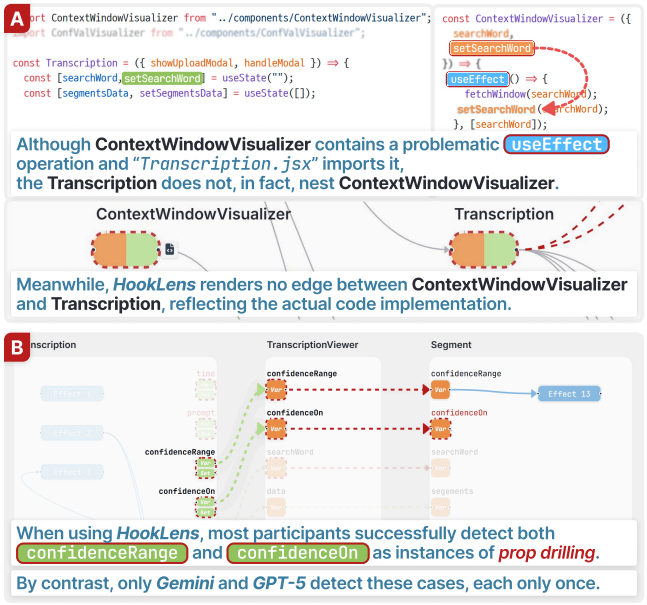

Figure 8: Falsely reported result within *Confides* project by LLM-based coding assistants. LLMs detect the code snippet as *effect modifying parent states*, while it does not produce anti-patterns because `Transcription Component` does note nest `ContextWindowVisualizer Component` in fact (A). Meanwhile, LLMs fail to detect *prop drilling* patterns although it is detected by most participants with HOOKLENS in the user study (B).

hook [33], as well as external state management systems like *Redux*[5]. These mechanisms are also essential for understanding the behavior of *React* applications, as they influence rendering processes in ways similar to `State Hooks`. Moreover, they enable data access across `Component` hierarchies, thereby mitigating anti-patterns such as *prop drilling*. Nevertheless, extending HOOKLENS to support these mechanisms introduces two major challenges.

First, the rule-based approach used in HOOKLENS requires substantial manual effort to expand its coverage, although it produces fast and consistent results. To address this limitation, LLM-based approaches could offer a promising alternative. Recent studies demonstrate the effectiveness of LLMs in both syntactic and semantic feature extraction from software [58, 20]. In our preliminary tests, we also observe that *GPT-5* shows potential for extracting `Components`, `States`, and `Effects` in a manner comparable to the rule-based extraction logic.

[5] https://redux.js.org/

Next, attempting to represent these mechanisms using the current node-link diagram in HOOKLENS may increase visual complexity by generating excessive nodes and edges or disrupting the layout. Future work should investigate alternative visualization methods to effectively represent them including other built-in `Hooks`. This challenge is common in node-link diagram representations, where various techniques (e.g., grid layout for grouped graph [68] or bubble set [8]) have been explored to address similar issues.

9.4 Visual Analytics in the AI Era

Our study suggests that developers, with the support of HOOKLENS, achieve a deeper understanding of code structures and detect anti-patterns more effectively than state-of-the-art LLM-based coding assistants (Section 8). However, our evaluation is conducted with limited prompt engineering and a constrained number of interaction rounds, and the strong performance of recent models such as *GPT-5* suggests that future coding assistants may increasingly understand broader code contexts and deliver better results. In fact, several recent studies and commercial systems propose frameworks and techniques that enable LLMs to retain and reason over richer contextual information [26, 46]. Despite these advances, prior studies [66, 42, 48] and our findings indicate that LLM-based assistants can still introduce basic errors or overlook subtle but critical dependencies. Therefore, visual analytics for software remains essential not only for helping developers comprehend complex projects but also for providing a foundation to interpret, monitor, and guide the behavior of LLM-based assistants in emerging coding paradigms. Future research should thus expand its focus beyond conventional programming environments to emerging ecosystems that integrate LLMs and other AI technologies. Recent studies such as *NeuroSync* [69] and *DreamGarden* [10] present promising directions by leveraging visualization to support users in understanding and controlling AI-driven code generation processes.

10 CONCLUSION

HOOKLENS offers a novel approach to addressing the challenges of understanding *React* applications and detecting anti-patterns. By visually representing the relationships between `Components` and `Hooks`, HOOKLENS enables developers to identify anti-patterns and inspect relative `Components` and `Hooks` along with their source code. Through preliminary interviews and an iterative design process, we develop HOOKLENS, an interactive visual analytics system for exploring *React* applications from the perspective of `Hooks` and anti-patterns. Findings from our user study and comparative evaluation show that HOOKLENS not only supports understanding code structures and detecting anti-patterns, but also highlights that such understanding remains essential even when using LLM-based coding assistants. Therefore, extending visual analytics to software frameworks such as *React* holds strong potential to help developers build and maintain interactive applications more effectively.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2023R1A2C200520911), the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) [NO.RS-2021-II211343, Artificial Intelligence Graduate School Program (Seoul National University)], and in part by Samsung Electronics. The ICT at Seoul National University provided research facilities for this study. Hyeon Jeon is in part supported by Google Ph.D. Fellowship.

References

[1] M. Almeida, G. Cole, K. Du, G. Luo, S. Pan, Y. Pan, K. Qiu, V. Reddy, H. Zhang, Y. Zhu, and C. Omar. Rustviz: Interactively visualizing ownership and borrowing. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–10, 2022. doi: 10.1109/VL/HCC53370.2022.9833121 3

[2] ANTHROPIC. Claude code: Your code's new collaborator. https://www.anthropic.com/claude-code, 2022. Accessed: 2025-09-10. 8

[3] I. Bacher, B. M. Namee, and J. D. Kelleher. On using tree visualisation techniques to support source code comprehension. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 91–95, 2016. doi: 10.1109/VISSOFT.2016.8 8

[4] H. Banken, E. Meijer, and G. Gousios. Debugging data flows in reactive programs. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, p. 752–763. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/3180155.3180156 3

[5] S. Boersma and M. Lungu. React-bratus: Visualising react component hierarchies. In *2021 Working Conference on Software Visualization (VISSOFT)*, pp. 130–134, 2021. doi: 10.1109/VISSOFT52517.2021.00025 3, 4

[6] K. Borowski, B. Balis, and T. Orzechowski. Graph buddy — an interactive code dependency browsing and visualization tool. In *2022 Working Conference on Software Visualization (VISSOFT)*, pp. 152–156, 2022. doi: 10.1109/VISSOFT55257.2022.00023 3, 8

[7] N. Chotisarn, L. Merino, X. Zheng, S. Lonapalawong, T. Zhang, M. Xu, and W. Chen. A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4):539–558, 2020. doi: 10.1007/s12650-020-00647-w 3

[8] C. Collins, G. Penn, and S. Carpendale. Bubble sets: Revealing set relations with isocontours over existing visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1009–1016, 2009. doi: 10.1109/TVCG.2009.122 9

[9] R. Du, N. Li, J. Jin, M. Carney, S. Miles, M. Kleiner, X. Yuan, Y. Zhang, A. Kulkarni, X. Liu, A. Sabie, S. Orts-Escolano, A. Kar, P. Yu, R. Iyengar, A. Kowdle, and A. Olwal. Rapsai: Accelerating machine learning prototyping of multimedia applications through visual programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, 2023. doi: 10.1145/3544548.3581338 3

[10] S. Earle, S. Parajuli, and A. Banburski-Fahey. Dreamgarden: A designer assistant for growing games from a single prompt. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25. Association for Computing Machinery, New York, NY, USA, 2025. doi: 10.1145/3706598.3714233 3, 9

[11] ESLint. Espree. https://github.com/eslint/js/blob/main/packages/espree/, 2024. Accessed: 2025-10-27. 5

[12] A. M. Fard and A. Mesbah. JSNOSE: Detecting JavaScript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 116–125, 2013. doi: 10.1109/SCAM.2013.6648192 1, 2

[13] F. Ferreira, H. S. Borges, and M. T. Valente. Refactoring React-based web apps. *Journal of Systems and Software*, 215:112105, 2024. doi: 10.1016/j.jss.2024.112105 2, 3

[14] F. Ferreira and M. T. Valente. Detecting code smells in React-based web apps. *Information and Software Technology*, 155:107111, 2023. doi: 10.1016/j.infsof.2022.107111 1, 2, 3

[15] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018. 4, 5

[16] Google. Gemini CLI: Build, debug & deploy with AI. https://geminicli.com/, 2025. Accessed: 2026-01-25. 8

[17] C. Gouveia, J. Campos, and R. Abreu. Using html5 visualizations in software fault localization. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 1–10, 2013. doi: 10.1109/VISSOFT.2013.6650539 3, 8

[18] Z. Guo, M. Kang, V. Venkatakrishnan, R. Gjomemo, and Y. Cao. Reactappscan: Mining react application vulnerabilities via component graph. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, p. 585–599. Association for Computing Machinery, New York, NY, USA, 2024. doi: 10.1145/3658644.3670331 3

[19] C. Han, J. Lieffers, C. Morrison, and K. E. Isaacs. An overview+detail layout for visualizing compound graphs. In *2024 IEEE Visualization and Visual Analytics (VIS)*, pp. 136–140, 2024. doi: 10.1109/VIS55277.2024.00035 8

[20] V. N. Ignatyev, N. V. Shimchik, D. D. Panov, and A. A. Mitrofanov. Large language models in source code static analysis. In *2024 Ivannikov Memorial Workshop (IVMEM)*, pp. 28–35, 2024. doi: 10.1109/IVMEM63006.2024.10659715 9

[21] T. Ishio, S. Etsuda, and K. Inoue. A lightweight visualization of interprocedural data-flow paths for source code reading. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pp. 37–46, 2012. doi: 10.1109/ICPC.2012.6240506 3, 8

[22] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. 8

[23] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012. doi: 10.1007/s10664-011-9171-y 4, 5

[24] Y. Kim, H. Jeon, Y.-H. Kim, Y. Ki, H. Song, and J. Seo. Visualization support for multi-criteria decision making in software issue propagation. In *2021 IEEE 14th Pacific Visualization Symposium (PacificVis)*, pp. 81–85, 2021. doi: 10.1109/PacificVis52677.2021.00018 3

[25] Y. Kim, J. Kim, H. Jeon, Y.-H. Kim, H. Song, B. Kim, and J. Seo. Githru: Visual analytics for understanding software development history through git metadata analysis. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):656–666, 2021. doi: 10.1109/TVCG.2020.3030414 3

[26] R. Krishna, R. Pan, S. Sinha, S. Tamilselvam, R. Pavuluri, and M. Vukovic. Codellm-devkit: A framework for contextualizing code llms with program analysis insights. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, p. 308–318, 2025. doi: 10.1145/3696630.3728555 9

[27] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 117–124, 2011. doi: 10.1109/VLHCC.2011.6070388 3

[28] J. Lee, J. Ahn, and K. Yi. React-trace: A semantics for understanding react hooks: An operational semantics and a visualizer for clarifying react hooks. *Proc. ACM Program. Lang.*, 9(OOPSLA2), Oct. 2025. doi: 10.1145/3763067 3

[29] Y. Lin, L. Yang, H. Li, H. Qu, and D. Moritz. Interlink: Linking text with code and output in computational notebooks. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, 2025. doi: 10.1145/3706598.3714104 3, 8

[30] K. Liu and S. Reddivari. Visual analytics in software maintenance: A systematic literature review. In *Proceedings of the 2023 ACM Southeast Conference*, ACMSE '23, p. 70–77, 2023. doi: 10.1145/3564746.3587022 3

[31] Y. Lu, A. Leung, A. Swearngin, J. Nichols, and T. Barik. Misty: Ui prototyping through interactive conceptual blending. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, 2025. doi: 10.1145/3706598.3713924 3

[32] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, Apr. 1986.

doi: 10.1145/22949.22950 5

[33] Meta Platforms, Inc. Built-in react hooks - react. https://react.dev/reference/react/hooks, 2024. Accessed: 2025-01-07. 2, 9

[34] Meta Platforms, Inc. Passing data deeply with context - react. https://react.dev/learn/passing-data-deeply-with-context, 2024. Accessed: 2025-01-07. 2

[35] Meta Platforms, Inc. Component - react. https://react.dev/reference/react/Component, 2025. Accessed: 2025-08-25. 2

[36] Meta Platforms, Inc. Lifecycle of reactive effects - react. https://react.dev/learn/lifecycle-of-reactive-effects, 2025. Accessed: 2025-10-30. 2

[37] Meta Platforms, Inc. Render and commit - react. https://react.dev/learn/render-and-commit, 2025. Accessed: 2025-10-30. 2

[38] Meta Platforms, Inc. State: A component's memory - react. https://react.dev/learn/state-a-components-memory, 2025. Accessed: 2025-09-05. 2

[39] Meta Platforms, Inc. Synchronizing with effects - react. https://react.dev/learn/synchronizing-with-effects, 2025. Accessed: 2025-09-05. 2

[40] Meta Platforms, Inc. You might not need an effect - react. https://react.dev/learn/you-might-not-need-an-effect, 2025. Accessed: 2025-01-07. 2

[41] Microsoft. Using the compiler api. https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API, 2023. Accessed: 2025-10-27. 5

[42] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz. Reading between the lines: Modeling user behavior and costs in ai-assisted programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24, 2024. doi: 10.1145/3613904.3641936 9

[43] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen. Detection of embedded code smells in dynamic web applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 282–285, 2012. doi: 10.1145/2351676.2351724 2

[44] M. Nunes, C. Bezerra, F. Ferreira, B. Gois, and M. T. Valente. Detection of code smells in react with typescript applications. *Information and Software Technology*, 187:107835, 2025. doi: 10.1016/j.infsof.2025.107835 3

[45] OpenAI. Codex CLI: Pair with Codex in your terminal. https://developers.openai.com/codex/cli, 2025. Accessed: 2025-09-10. 8

[46] Oraios AI. About serena. https://oraios.github.io/serena/01-about/000_intro.html, 2025. Accessed: 2026-01-04. 9

[47] S. Park, S. Lee, E. Choi, H. Kim, M. Kweon, Y. Song, and J. Seo. Bridging gulfs in ui generation through semantic guidance, 2026. doi: 10.48550/arXiv.2601.19171 3

[48] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Commun. ACM*, 68(2):96–105, Jan. 2025. doi: 10.1145/3610721 9

[49] J. Qiu. *React Anti-Patterns: Build efficient and maintainable React applications with test-driven development and refactoring*. Packt Publishing Ltd, 2024. 2

[50] B. Saket, P. Simonetto, S. Kobourov, and K. Börner. Node, node-link, and node-link-group diagrams: An evaluation. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2231–2240, 2014. doi: 10.1109/TVCG.2014.2346422 3, 5

[51] G. Salvaneschi and M. Mezini. Debugging for reactive programming. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, p. 796–807. Association for Computing Machinery, New York, NY, USA, 2016. doi: 10.1145/2884781.2884815 3

[52] J. Sauro and J. R. Lewis. *Quantifying the user experience: Practical statistics for user research*. Morgan Kaufmann, 2016. 6

[53] T. Sharma, S. Gupta, and U. R. Singh. Analyzing the difference between ReactJS and AngularJS. In *2023 International Conference on Computational Intelligence, Communication Technology and Networking (CICTN)*, pp. 37–42, 2023. doi: 10.1109/CICTN57981.2023.10141276 3

[54] A. Shatnawi, H. Mili, G. El Boussaidi, A. Boubaker, Y.-G. Guéhéneuc, N. Moha, J. Privat, and M. Abdellatif. Analyzing program dependencies in java ee applications. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 64–74, 2017. doi: 10.1109/MSR.2017.6 3, 8

[55] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pp. 336–343, 1996. doi: 10.1109/VL.1996.545307 5

[56] L. L. Silva, J. R. d. Silva, J. E. Montandon, M. Andrade, and M. T. Valente. Detecting code smells using chatgpt: Initial insights. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '24, p. 400–406, 2024. doi: 10.1145/3674805.3690742 8

[57] stackoverflow. 2024 developer survey - most popular technologies. https://survey.stackoverflow.co/2024/technology, 2024. 2, 6

[58] A. A. T. Talukder, O. Alam, and A. Azim. Leveraging llms for automatic feature extraction in embedded systems to support software reuse. In *2025 IEEE International Conference on Information Reuse and Integration and Data Science (IRI)*, pp. 1–6, 2025. doi: 10.1109/IRI66576.2025.00009 9

[59] H. Tarner, D. van den Bongard, and F. Beck. Visually analyzing the structure and code quality of component-based web applications. In *2021 Working Conference on Software Visualization (VISSOFT)*, pp. 160–164, 2021. doi: 10.1109/VISSOFT52517.2021.00031 3, 4

[60] C. Tessa, M. Bochicchio, and F. A. Fontana. Exploring architectural smells detection through llms. In V. Andrikopoulos, C. Pautasso, N. Ali, J. Soldani, and X. Xu, eds., *Software Architecture*, pp. 90–98, 2026. doi: 10.1007/978-3-032-02138-0_6 8

[61] C. Tseng, G. J. Quadri, Z. Wang, and D. A. Szafir. Measuring categorical perception in color-coded scatterplots. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23. Association for Computing Machinery, New York, NY, USA, 2023. doi: 10.1145/3544548.3581416 5

[62] A. Turcotte, M. D. Shah, M. W. Aldrich, and F. Tip. Drasync: identifying and visualizing anti-patterns in asynchronous javascript. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, p. 774–785. Association for Computing Machinery, New York, NY, USA, 2022. doi: 10.1145/3510003.3510097 5

[63] vdslab. Papervis. https://github.com/vdslab/papers_vis, 2022. Accessed: commit `2cc206f`. 6

[64] E. Visser. Meta-programming with concrete object syntax. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, GPCE '02, p. 299–315. Springer-Verlag, Berlin, Heidelberg, 2002. doi: 10.1007/3-540-45821-2_19 5

[65] washuvis. Confides: A Visual Analytics Solution for Automated Speech Recognition Analysis and Exploration. https://github.com/washuvis/vis2024confides, 2024. Accessed: commit `275e235`. 6

[66] J. D. Weisz, S. V. Kumar, M. Muller, K.-E. Browne, A. Goldberg, K. E. Heintze, and S. Bajpai. Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, CHI EA '25, 2025. doi: 10.1145/3706599.3706670 9

[67] D. Wu, F. Mu, L. Shi, Z. Guo, K. Liu, W. Zhuang, Y. Zhong, and L. Zhang. ismell: Assembling llms with expert toolsets for code smell detection and refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, p. 1345–1357, 2024. doi: 10.1145/3691620.3695508 8

[68] V. Yoghourdjian, T. Dwyer, G. Gange, S. Kieffer, K. Klein, and K. Marriott. High-quality ultra-compact grid layout of grouped networks. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):339–348, 2016. doi: 10.1109/TVCG.2015.2467251 9

[69] W. Zhang, L. Shen, S. Xu, J. Wang, J. Zhao, H. Qu, and L.-P. Yuan. Neurosync: Intent-aware code-based problem solving via direct llm understanding modification. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*, UIST '25, 2025. doi: 10.1145/3746059.3747668 3, 9