Automated Detection of Architectural Anti-Patterns in React Applications Using Static Analysis

Pragnya Ambekar

A thesis submitted in partial fulfillment of the

requirements for the degree of

Masters of Science in Computer Science and Software Engineering

University of Washington

2026

Committee:

Dr. Min Chen
Dr. Kelvin Sung
Dr. Clark Olson

Program Authorized to Offer Degree:

Computer Science and Engineering

University of Washington

Abstract

Automated Detection of Architectural Anti-Patterns in React Applications Using Static Analysis

Pragnya Ambekar

Chair of the Supervisory Committee:
Dr. Min Chen
Department of Computing and Software Systems

This research provides a reusable framework for measuring architectural quality in React applications, transforming qualitative architectural principles into measurable, automated checks. The tool processes codebases at speeds suitable for continuous integration pipeline integration and produces severity-tiered output that guides development teams in prioritizing refactoring effort. The validation methodology, combining stratified sampling, independent expert review, and pre-defined outcome mapping, provides a template applicable to future code smell detection research in React and other frameworks.

A systematic comparison with SniffTSX, a recently published React smell detector focused on TypeScript type safety, reveals minimal overlap between the two tools. Of twelve combined pattern categories, only two show any partial overlap, confirming that the tools address orthogonal concerns and complement rather than compete with each other. SniffTSX ensures TypeScript correctness and React API compliance; this tool ensures architectural soundness. Together they provide comprehensive quality coverage across dimensions that neither addresses alone.

The tool was evaluated against four open-source repositories: Grafana, Mattermost, Refine, and TodoMVC. It processed 8,458 components across approximately 2.3 million lines of code at an average rate of 452 components per minute, producing 544 detections with a consistent detection rate of 5.4 to 6.9 percent across all four repositories. A formal validation study involving three independent reviewers who assessed 71 stratified components achieved 100 percent completion and an average precision of 86.39 percent across all reviewers. God Component detection, the primary contribution, achieved 98.04 percent precision with only one false positive out of 51 assessed components. Extreme severity cases achieved 100 percent precision, and Severe severity cases achieved 96.43 percent precision. Average pairwise inter-rater reliability was 62.5 percent, reflecting the inherent subjectivity of architectural quality assessment.

This thesis presents a pattern detection tool that identifies six React-specific architectural anti-patterns through Abstract Syntax Tree analysis and combined metric thresholds. The six patterns are: God Components, Prop Drilling, Inline Functions, Duplicate State, Deep JSX Nesting, and useEffect

Dependency Issues. Each pattern is measured using quantifiable code metrics including Lines of Code, JSX element count, and hook count, then classified across three severity tiers, Extreme, Severe, and Moderate, using thresholds calibrated from real production codebases. The primary methodological contribution is the combined metric approach for God Component detection, which requires a component to simultaneously exceed thresholds on all three metrics rather than any single one, substantially reducing false positives relative to single-metric detection.

This research addresses the problem of detecting architectural anti-patterns in React applications through automated static code analysis. While React's component-based architecture simplifies user interface development, certain structural inefficiencies emerge gradually over time: components that accumulate too many responsibilities, props passed through excessive levels of the component hierarchy, deeply nested rendering logic, and incorrectly declared hook dependencies. These patterns do not produce immediate failures, but they progressively degrade maintainability, increase the cost of future changes, and introduce subtle bugs that are difficult to trace.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my committee chair, Professor Min Chen, for her guidance, mentorship, and continued support throughout this research. Her insights fundamentally shaped the direction of this work, from the initial problem formulation through to the final validation study design.

I am deeply grateful to the three independent reviewers who participated in the validation study. Their time, expertise, and careful assessment of 71 React components formed the empirical foundation of this research. Without their contribution the core findings of this thesis would not have been possible.

TABLE OF CONTENTS

Chapter 1: Introduction

1.1 Background and Motivation

Before examining the specific problem this thesis addresses, it is worth establishing what the term architectural anti-pattern means and why it matters in the context of software development. The concept was formally introduced by Brown and colleagues in 1998 as a systematic counterpart to the design patterns catalog popularized by the Gang of Four. Where a design pattern captures a proven, reusable solution to a recurring design problem, an anti-pattern captures something equally recurring but harmful: a structural arrangement that developers repeatedly reach for because it seems like a reasonable approach, but that consistently produces negative consequences when examined over time.

What makes anti-patterns distinct from ordinary bugs is that they are not mistakes in the conventional sense. Code containing an anti-pattern compiles without errors, passes its test suite, and behaves exactly as intended at the moment it is written. The damage is not immediate, it is structural and cumulative. Each instance of an anti-pattern makes the codebase slightly harder to understand, slightly more expensive to modify, and slightly more likely to introduce defects during future changes. In component-based frontend systems like React, this structural damage manifests at the component level: in how responsibilities are divided across components, how data travels through the component hierarchy, and how state is declared and shared. A component built around an anti-pattern today is a component that will slow down the next developer who needs to change it, test it, or build on top of it.

This distinction between code that is wrong and code that is structurally problematic is central to the research problem this thesis investigates. Existing automated tools are well-equipped to detect wrongness. They are poorly equipped to detect structural harm that accumulates gradually and only becomes visible at scale.

Large-scale frontend systems present a category of architectural challenges that is distinct from those encountered in backend or systems programming. Unlike a service that processes requests and returns responses, a frontend application is composed of a hierarchy of interactive components that each carry their own state, rendering logic, and side effects. React, the most widely adopted framework for building these systems, simplifies the initial construction of component-based user interfaces by providing a declarative programming model and a clear separation between component state and rendered output. This simplicity is one of React's greatest strengths, but it also contributes to a well-documented long-term problem: the same flexibility that makes React easy to start building with also makes it easy to build systems that gradually degrade in architectural quality as they grow.

The mechanism by which this degradation occurs is well understood by experienced React practitioners, even if it is rarely discussed in formal research literature. A component begins as a focused, coherent unit that handles a single concern, perhaps rendering a user profile card or managing a simple form. Over the months that follow, product requirements evolve. The profile card needs to support an editing mode. The form needs validation logic, loading states, and error handling. An API call gets added, then a second one. A new design requirement introduces conditional rendering that depends on multiple pieces of state. Each of these changes is individually reasonable and locally justifiable. None of them, considered in isolation, constitutes a design failure. But collectively, they transform the original focused component into

something qualitatively different: a component that manages data fetching, business logic, UI rendering, error states, and user interaction in a single 400-line function with twelve hooks and dozens of JSX elements. The team knows the component is problematic, but at no point during its growth did a clear line get crossed that an automated tool could have flagged.

This incremental accumulation of architectural debt is the central problem that motivated this research. It is different in character from the kinds of problems that existing code quality tools address. ESLint operates at the syntactic level, detecting violations of coding conventions such as unused variables, missing semicolons, and incorrect usage of language features. These are problems that have clear right answers independent of context. TypeScript operates at the semantic level, detecting type errors that would cause incorrect behavior at runtime. These too have clear right answers: a function that expects a string and receives a number is wrong regardless of the purpose it serves. SonarQube and similar general-purpose static analysis platforms flag a broader range of concerns including code complexity, duplication, and certain design anti-patterns, but they do so using metrics originally developed for object-oriented Java codebases and lack the framework-specific awareness needed to understand React's component model, hook system, or JSX rendering patterns.

The consequence of this tool landscape gap is that React development teams face a specific category of quality problem, architectural anti-patterns in component design, for which no automated detection exists. A developer can run ESLint, TypeScript, and SonarQube against a React codebase and receive a clean report, while the codebase simultaneously contains dozens of God Components that mix multiple responsibilities, deep prop drilling chains that couple distant components together, and hook dependency arrays that produce subtle bugs through stale closures. These problems are invisible to all three tools because none of them are designed to reason about React component architecture.

The scale of this problem is substantial. React remains the most widely used frontend framework in the professional software development community. According to the Stack Overflow 2024 Developer Survey, 40.6 percent of professional developers use React, a proportion that has remained stable or grown for several consecutive years. This means that the architectural quality of React applications is a concern that affects a significant fraction of all production software currently being built and maintained. A study conducted in 2023 surveying 500 React developers found that 68 percent reported spending significant time refactoring components that had grown too complex, and 52 percent indicated that their teams lacked clear guidelines for when a component should be decomposed. These figures suggest that the problem is not an edge case experienced by a small minority of teams but a widespread, ongoing challenge that the community lacks adequate tools to address.

The absence of automated architectural detection has a compounding effect on team dynamics. Senior developers who have internalized React architectural principles can recognize problematic components through code review, but code review is inconsistent, does not scale with team size, and depends entirely on the reviewer's experience and attention at the time of review. Junior developers, who have not yet developed the architectural intuition that comes from years of maintaining large React codebases, lack the concrete guidance needed to recognize when their components are crossing from acceptable complexity into problematic territory. The result is an asymmetry in code quality contribution that is not addressed by any existing tooling: senior developers spend disproportionate review effort on architectural concerns that could be screened automatically, while junior developers receive inconsistent and often subjectively framed feedback that does not help them develop lasting architectural judgment.

This research addresses the gap directly by building, calibrating, and validating an automated tool for detecting architectural anti-patterns in React applications. The tool operates through static analysis of TypeScript and JSX source code using the TypeScript Compiler API, extracts quantitative metrics from Abstract Syntax Tree representations of each component, and applies empirically calibrated threshold combinations to classify components by the severity of the architectural concerns they exhibit. By transforming qualitative architectural judgments into measurable, automated checks, the tool provides the concrete and consistent feedback that both individual developers and development teams currently lack.

1.2 Problem Statement

The central problem this research addresses can be stated precisely: React development teams lack automated tools capable of detecting architectural anti-patterns in component design before those patterns accumulate into technical debt that requires substantial effort to remediate. This problem has three distinct dimensions that together explain both why it exists and why it has not been adequately addressed by prior work.

The first dimension is the detection gap in the existing tool landscape. The tools available to React development teams address correctness and syntactic quality but not architectural quality. Table 1.1 characterizes the four principal categories of automated quality tool available to React developers and the aspect of code quality each addresses.

Table 1.1: Automated quality tool categories and their coverage

Tool Category	Example	What It Checks	What It Cannot Check
Linters	ESLint	Syntax violations, coding conventions, potential runtime errors	Component size, responsibility boundaries, architectural structure
Type Checkers	TypeScript	Type correctness, API contract compliance	Whether components violate single responsibility, prop drilling depth
General Static Analysis	SonarQube	Cyclomatic complexity, code duplication, general smells	React-specific patterns, hook usage, JSX structural complexity
React-Specific Tools	SniffTSX	TypeScript type safety, React API correctness	Component architecture, God Components, structural anti-patterns

Each tool in this landscape addresses a well-defined category of concern with demonstrable effectiveness within that category. The gap is not that these tools are inadequate at what they do; it is that none of them does what architectural pattern detection requires. A codebase that passes all four categories of tool check

can still contain dozens of architecturally problematic components, and no existing tool would report any of them.

The second dimension is the gradual and threshold-free nature of architectural degradation. Unlike type errors or syntax violations, architectural anti-patterns do not emerge at a single identifiable moment. A component does not become a God Component when a specific line of code is written; it becomes one through the accumulation of many individually reasonable additions over an extended period. This gradual nature means that human-driven processes such as code review are poorly suited to catching architectural problems at the point where they first become concerning. By the time a component is clearly large enough to attract a code review comment, it has often already grown beyond the point where decomposition is straightforward and low-risk. Automated detection that monitors component complexity continuously and flags components as they cross defined thresholds addresses this temporal dimension in a way that periodic code review cannot.

The third dimension is the knowledge asymmetry between experienced and inexperienced React developers. Experienced developers have internalized architectural principles through years of maintaining large codebases and have developed intuitions about when components are growing too complex. They can recognize a God Component by reading a few lines of its function signature and state declarations. Junior developers have not had these experiences and cannot reliably apply architectural principles that are described only in qualitative terms. Telling a junior developer to keep components small and focused is advice that they cannot operationalize without knowing what small means in measurable terms. Automated tools that express architectural principles as concrete, measurable thresholds provide the concrete guidance that qualitative advice cannot.

The economic dimension of this problem is also worth stating explicitly. Research on the cost of technical debt consistently finds that the cost of remediating a defect grows substantially with the time elapsed between its introduction and its detection. For architectural anti-patterns, this cost growth is particularly steep because architectural refactoring requires coordinating changes across multiple files, updating tests, and validating that the decomposed components preserve the behavior of the original. Studies in software maintenance economics have estimated that for every dollar spent preventing technical debt at the time of introduction, teams save between four and ten dollars in future remediation costs. The absence of automated architectural detection means that React teams are systematically spending in the more expensive remediation category when early detection tools could shift that cost to the cheaper prevention category.

1.3 Research Questions

This research is structured around three questions that together span the feasibility, characteristics, and practical positioning of automated React architectural detection.

RQ1: Can React architectural anti-patterns be reliably detected using combined metric thresholds and AST traversal, and what precision can be achieved compared to human expert review?

This question addresses the most fundamental feasibility concern: whether the class of problems described in Section 1.2 is amenable to automated static analysis at all. Architectural quality is widely understood to involve contextual judgment that resists simple rule-based capture, and prior work in code smell detection for object-oriented languages has repeatedly found that single-metric approaches produce

false positive rates too high for practical use. The question therefore has two parts: first, whether static analysis can identify architectural anti-patterns at all, and second, whether the precision achievable is high enough for the tool's output to be trusted by developers in practice.

The question cannot be answered by theoretical argument alone. It requires empirical validation against the judgments of experienced developers who assess the same components that the tool flags, without knowledge of the tool's classifications, and provide independent ground truth against which precision can be measured. The validation study described in Chapters 5 and 6 was designed to answer this question rigorously, using three independent reviewers, 71 stratified components, and a pre-defined outcome mapping that defines what constitutes a true positive and a false positive before any reviews are conducted.

RQ2: How do detected anti-patterns correlate with codebase characteristics, and do larger or older codebases exhibit more anti-patterns than smaller or newer ones?

This question examines the distribution and prevalence of architectural anti-patterns across codebases that differ substantially in scale, age, and development context. The question has both practical and theoretical significance. Practically, understanding which patterns dominate across different types of codebases informs where development teams should focus refactoring effort. If God Components account for nearly half of all detections across all repository types, as this research finds, then component decomposition should be the primary architectural investment for most teams regardless of their codebase characteristics.

Theoretically, the question probes the nature of technical debt accumulation in React codebases. The conventional model of technical debt suggests that it accumulates as a function of codebase age and size, with older and larger codebases exhibiting more problems than younger and smaller ones. If this model is correct, detection rates should be substantially higher in Grafana and Mattermost, both mature codebases over a decade old, than in Refine and TodoMVC. If detection rates are similar across all four repositories, the data would suggest a more dynamic model in which debt accumulates through feature development and is continuously offset through refactoring, producing a roughly stable equilibrium level of architectural complexity.

RQ3: How does this tool compare to existing React analysis tools, and does it provide complementary value or duplicate existing capabilities?

This question addresses the positioning of the tool within the broader landscape of React code quality tooling described in Section 1.2. Even a highly accurate architectural detection tool would provide limited value if it substantially overlapped with existing tools, because teams already using those tools would gain little by adding another. The comparison with SniffTSX is particularly important because SniffTSX is the most recent and most directly comparable React-specific quality tool, and superficial descriptions of both tools as React code quality analyzers might suggest they address the same concerns.

The answer to this question determines the practical recommendation for how teams should deploy the two tools: as alternatives where one suffices, or as complements where both are needed to achieve comprehensive coverage. As demonstrated in Chapter 7, the answer is unambiguously complementary, with the two tools addressing orthogonal concerns with minimal pattern overlap, but establishing this empirically rather than asserting it requires the systematic pattern-by-pattern comparison that this research provides.

1.4 Thesis Contributions

This thesis makes five contributions to React code quality research and tooling, each of which addresses one or more of the gaps identified in Section 1.2 and answers one or more of the research questions posed in Section 1.3. Table 1.2 summarizes the five contributions, the research question each addresses, and the primary evidence supporting each claim.

Table 1.2: Thesis contributions, research questions addressed, and supporting evidence

Contribution	**Description**	**Research Question**	**Primary Evidence**
Pattern Detection Framework	AST-based detection system for 6 React-specific architectural anti-patterns, implemented as a configurable command-line tool	RQ1, RQ3	544 detections across 8,458 components in 4 repositories
Combined Metric Approach	God Component detection using simultaneous LOC, JSX element count, and hook count thresholds, achieving 98.04% precision	RQ1	Formal validation study: 50 true positives out of 51 assessed God Components
Empirical Validation Study	Three independent reviewers assessed 71 stratified components, achieving 86.39% average precision and 100% completion rate	RQ1	Chapter 6 validation results
Tool Comparison Study	First systematic comparison between architectural and type-safety focused React tools, demonstrating complementary rather than competing coverage	RQ3	Pattern overlap analysis: only 2 of 12 combined pattern categories show any overlap
Production-Scale Evaluation	Analysis of 8,458 components across 4 real-world codebases at 452 components per minute with consistent 5.4	RQ2	Chapter 6 detection results across all four repositories

Contribution	Description	Research Question	Primary Evidence
	to 6.9 percent detection rates		

The pattern detection framework is the foundational technical contribution. It implements a modular pipeline architecture in which each of the six detector modules operates independently on the Abstract Syntax Tree produced by the TypeScript Compiler API, allowing patterns to be added or removed without modifying existing detection logic. The framework is designed to be extensible to additional React patterns and adaptable to other component-based frameworks such as Vue or Angular. The framework is designed to be extensible to additional React patterns and adaptable to other component-based frameworks. The architectural principles underlying God Component detection, Prop Drilling analysis, and JSX structural assessment are not React-specific in origin. They derive from general software engineering principles around single responsibility, coupling, and cohesion that apply wherever software is organized into composable, reusable units. The specific implementation details are React-specific, but the detection methodology is not. This distinction is discussed in Chapter 8 in the context of framework generalization as a future research direction.

The combined metric approach for God Component detection is the core methodological contribution and the most significant finding of this work. Prior approaches to component complexity assessment relied on single metrics, most commonly Lines of Code, with estimated precision in the range of 60 to 70 percent. The insight underlying the combined approach is that architectural problems are multidimensional: a component violates single responsibility not merely by being large in terms of code volume, but by being simultaneously large in code volume, complex in rendered structure, and heavy in state management. Requiring all three conditions to be satisfied simultaneously filters out the false positives that arise when any one metric is elevated for legitimate reasons, producing 98.04 percent precision for God Component detection in the formal validation study.

The empirical validation study makes a methodological contribution as well as an empirical one. The study design, including stratified sampling across severity tiers, reviewer independence maintained through blind assessment, an assessment scale deliberately decoupled from the tool's severity classification, and a priori outcome mapping, addresses the most significant internal validity threats in code smell detection validation research. The use of three independent reviewers provides a more stable precision estimate and enables pairwise inter-rater reliability to be calculated across three independent reviewer pairs. The 86.39 percent average precision result and the precision gradient from 100 percent at Extreme through 96.43 percent at Severe to 55 percent at Moderate together provide a nuanced characterization of where the tool performs strongly and where human judgment remains essential.

The tool comparison study establishes a conceptual framework for thinking about React code quality as a multidimensional property requiring multiple specialized tools, rather than a single property that one comprehensive tool could in principle address. By demonstrating empirically that SniffTSX and this tool address distinct concerns with minimal overlap, the comparison provides the evidence base for a specific practical recommendation: development teams should use both tools in combination to achieve coverage of the full quality space that neither tool provides alone.

The production-scale evaluation provides evidence of generalizability that single-repository studies cannot. The consistency of detection rates between 5.4 and 6.9 percent across four repositories spanning three orders of magnitude in codebase size and ranging from 2013 to 2021 in age demonstrates that the calibrated thresholds capture a property of React codebases that holds across diverse development contexts. The 452 components per minute processing speed demonstrates practical deployability in the continuous integration contexts where the tool would provide the most value.

Having described the five contributions individually, it is worth stating the novelty of this work directly and in aggregate, since the significance of each contribution is best understood in relation to what existed before it.

Prior to this thesis, the React code quality research community had produced two relevant bodies of work. ReactSniffer established that React-specific architectural smells exist and are prevalent in production codebases, but it did not validate the precision of its detections through independent expert review and relied on file-level rather than component-level analysis. SniffTSX introduced rigorous precision validation to React-specific quality research, but it addressed type safety and API correctness rather than architectural quality, leaving the structural dimension of component design unexamined.

This thesis is the first work to address all three of the following simultaneously: automated detection of React architectural anti-patterns at the component level, empirical precision validation through independent expert review, and a systematic comparison with existing tools to establish complementary rather than duplicated value. No prior work has done all three together. ReactSniffer addressed the first without the second. SniffTSX addressed the second without the first. Neither addressed the third.

The specific methodological novelty is the combined metric approach. The idea of measuring component complexity is not new. What is new is the empirical demonstration that requiring simultaneous elevation across three independent metrics produces qualitatively better precision than any single-metric approach, and that this improvement is large enough to be practically meaningful: from approximately 62 percent for Lines of Code alone to 98.04 percent for the combined threshold. This finding is not a marginal improvement in an existing technique. It establishes a different detection principle, one grounded in the observation that genuine architectural violations are inherently multidimensional and that any detection approach that treats them as one-dimensional will systematically produce false positives.

The empirical novelty is the finding that architectural debt rates are consistent across codebases of very different ages and sizes. This result was not predicted by the conventional model of technical debt accumulation and has not been reported in prior React-specific quality research. It suggests that the relationship between codebase maturity and architectural quality is more dynamic than the literature has assumed, and it carries implications for how architectural monitoring should be integrated into development practice.

Chapter 2: Related Work

2.1 Code Smells and Anti-Patterns

The concept of code smells was introduced by Fowler in his 1999 book Refactoring: Improving the Design of Existing Code, where he described them as surface indicators of deeper design problems in software systems. The term is deliberately metaphorical: a smell does not prove that something is wrong in the way a compiler error does, but it signals that a closer look is warranted. Code smells do not prevent software from functioning; they compromise its maintainability, readability, and capacity for long-term evolution. A system full of code smells continues to deliver correct output, but it becomes progressively harder to understand, modify, and extend as it grows. (Fowler, 1999)

It is worth drawing a precise boundary between two related but distinct concepts: code smells and anti-patterns. Fowler used the term code smell to describe surface-level symptoms in source code that suggest something might be wrong heuristics that warrant closer inspection rather than definitive diagnoses. A long method or a large class is a code smell: it may indicate a problem, or it may be perfectly acceptable given its context. An anti-pattern, by contrast, carries a stronger claim. The term as used by Brown and colleagues implies a documented pattern of practice that is not merely suspicious but reliably harmful, one for which the negative consequences are understood and for which better alternatives have been identified. Every anti-pattern has a refactored solution; the harm lies in not recognizing that the problematic pattern has been adopted in the first place.

Throughout this thesis, the term anti-pattern is used in this stronger sense. The six patterns detected by the tool are not merely suspicious structural arrangements that might warrant investigation. They are well-characterized failure modes in React component design, each with known consequences for maintainability and each with established refactoring strategies. God Components should be decomposed. Prop Drilling should be resolved through context or state management. Inline functions in JSX attributes should be extracted or memoized. The automated detection this thesis provides is valuable precisely because these failure modes are identifiable from source code structure before their consequences become costly enough to demand attention.

The research literature that followed Fowler's original formulation established several important empirical findings about code smell prevalence and impact. Studies consistently find that code smells appear in a substantial fraction of classes in mature object-oriented codebases, with estimates ranging from 60 to over 80 percent depending on the smell types and detection criteria used. More significantly, empirical studies have found that classes containing code smells exhibit higher defect density, require more effort to modify, and are more likely to be the subject of bug reports than smell-free classes. These findings establish the practical importance of smell detection as a code quality activity: reducing smells is not merely an aesthetic concern but a measurable predictor of reduced maintenance cost and defect frequency. (Fowler, 1999)

The foundational work on quantitative code smell detection was established by Chidamber and Kemerer in their 1994 paper introducing the CK metrics suite. The six metrics they defined, Weighted Methods per Class, Depth of Inheritance Tree, Number of Children, Coupling Between Objects, Response for Class, and Lack of Cohesion of Methods, provided the first systematic framework for measuring object-oriented

design quality through quantifiable properties of source code. These metrics were not designed to detect specific smells but rather to quantify the structural properties that underlie many common smells: excessive coupling, deep inheritance hierarchies, and insufficient cohesion. Their work established the foundational principle, directly relevant to this thesis, that qualitative architectural concerns can be operationalized as measurable code properties that lend themselves to automated detection. (Chidamber & Kemerer, 1994)

McCabe's cyclomatic complexity metric, introduced in 1976, predates the CK suite and represents a different approach to code quality measurement. Rather than measuring structural relationships between classes, cyclomatic complexity measures the internal complexity of an individual function or method by counting the number of linearly independent paths through its control flow graph. A function with no branches has cyclomatic complexity of one. Each conditional statement, loop, or logical operator adds one to the count. McCabe established that cyclomatic complexity correlates with testing difficulty and defect rates, and his metric became widely adopted as a proxy for code quality in procedural and later object-oriented systems. Although cyclomatic complexity was designed for procedural code, it applies to React functional components as well and was considered as a candidate metric during the pattern selection process described in Chapter 3. It was ultimately not included in the combined God Component threshold because it measures internal control flow complexity rather than the multi-dimensional architectural complexity that distinguishes a God Component from a legitimately complex but well-organized function. (McCabe, 1976)

The earliest automated smell detection systems applied threshold-based rules to CK metrics and Lines of Code measurements. A class was flagged as a God Class if its Weighted Methods per Class exceeded a threshold, as a Data Class if it had few methods relative to its fields, and so on. These systems were effective for the smells they targeted but produced high false positive rates because single-metric thresholds are inherently imprecise. A class with high Weighted Methods per Class might be a legitimate utility class or a cohesive service object rather than a true God Class. This false positive problem motivated the investigation of combined metrics that this thesis addresses in the React context.

Machine learning approaches to code smell detection emerged as an alternative to rule-based systems in the 2010s. A systematic literature review by Mishra et al. analyzing 86 studies published between 2014 and 2024 documented a broad shift from rule-based detection toward supervised learning classifiers trained on labeled smell instances. Machine learning approaches offer the potential to learn subtle, multi-feature patterns that are difficult to encode in explicit rules, and several studies demonstrated precision and recall improvements over rule-based baselines on specific smell types and datasets. (Mishra et al., 2024)

However, Mishra et al. also documented persistent challenges that have limited the practical impact of machine learning approaches. Dataset imbalance, meaning that smell-positive instances are substantially rarer than smell-negative instances in typical codebases, creates biased classifiers that perform well on the majority class but poorly on the minority class of interest. Inconsistent ground truth annotation, where different labeling studies use different criteria for what constitutes a genuine smell instance, makes cross-study comparisons unreliable and limits the accumulation of reusable labeled datasets. Feature engineering choices, including which metrics to compute and how to normalize them across codebases of different sizes, have been found to affect classification accuracy more significantly than algorithm selection. (Mishra et al., 2024)

Al-Shaaby et al. conducted a more focused systematic literature review specifically examining machine learning approaches to bad smell detection, analyzing 50 primary studies. Their review found that Support Vector Machines and Random Forests were the most commonly applied algorithms, but confirmed that no single algorithm consistently outperformed others across all smell types and datasets. Their most practically significant finding was that dataset quality and feature engineering choices were more determinative of detection accuracy than algorithm selection, a conclusion that motivates this thesis's focus on carefully defined combined metrics rather than algorithmic sophistication. (Al-Shaaby et al., 2020)

The machine learning literature on code smell detection is directly relevant to this thesis in two respects. First, the finding that single metrics are insufficient and that multi-feature approaches are necessary to achieve reliable detection is corroborated by this thesis's empirical finding that Lines of Code alone achieves approximately 62 percent precision for God Component detection while the combined LOC, JSX, and hook count threshold achieves 98.04 percent. Second, the labeled dataset of 71 components with assessments from three independent reviewers produced by this thesis's validation study represents a foundation for future machine learning approaches to React architectural detection, as discussed in Section 8.3.

The broader theoretical context for this thesis's approach is the distinction between correctness smells, which describe code that is likely to produce incorrect behavior, and architectural smells, which describe code that is structurally sound but organizationally problematic. Correctness smells such as null pointer dereferences and resource leaks have clear objective definitions that lend themselves to high-precision automated detection. Architectural smells such as God Classes, Feature Envy, and Shotgun Surgery involve qualitative judgments about responsibility boundaries and design intent that are context-dependent and inherently more subjective. The 62.5 percent inter-rater reliability observed in this thesis's validation study confirms that architectural quality assessment involves genuine subjective judgment, which directly informs how the tool is positioned: as a screening mechanism that surfaces architectural concerns for human review rather than an autonomous arbiter of architectural quality.

2.2 Static Analysis for Code Quality

Static analysis is the examination of source code without executing it, with the goal of identifying properties of the program that would otherwise require testing, review, or runtime observation to discover. The range of properties that static analysis can detect spans from syntactic violations that any competent parser can identify, through semantic violations that require type inference and dataflow analysis, to structural properties that require reasoning about the overall organization of a program. Modern software development pipelines typically employ static analysis tools at multiple levels of this spectrum, with different tools targeting different categories of concern.

At the syntactic level, linters such as ESLint for JavaScript and TypeScript detect violations of coding conventions and patterns that are statistically associated with bugs. ESLint operates by matching source code against a configurable set of rules, each of which identifies a specific syntactic pattern such as an unused variable, a missing semicolon, or an equality comparison using double equals rather than triple equals. ESLint is highly effective at what it does because its rules target properties that have clear right answers independent of context. However, ESLint's rule-based model is fundamentally limited to properties that can be identified by examining individual syntactic constructs in isolation. It cannot detect

properties that require reasoning about the relationship between multiple components, the flow of data through a component hierarchy, or the cumulative effect of many individually acceptable decisions on the overall structure of a system. All six patterns detected by this tool fall into the category of properties that ESLint cannot address.

At the semantic level, TypeScript's type checker adds a layer of static analysis that reasons about the types of values flowing through a program. Type checking can detect a category of errors that linters cannot: incorrect assumptions about what kind of value a variable or function parameter holds. TypeScript's type checker transforms potential runtime errors into compile-time errors that are detected before the code is executed. Like linting, type checking is highly effective within its domain because type correctness is a property that has a definite answer for any given program: either the types are consistent or they are not.

The key observation for this thesis is that type safety and architectural quality are orthogonal concerns. A React codebase can be fully type-safe, with every variable, parameter, and return value correctly annotated and verified by the TypeScript compiler, while simultaneously containing God Components with hundreds of lines and dozens of hooks, deep prop drilling chains that tightly couple distant components, and inline functions that break memoization on every render. TypeScript's type checker has no concept of component size, responsibility distribution, or structural complexity. A 1,000-line component with 20 hooks and 75 JSX elements that is also perfectly type-safe would pass TypeScript's checks without any indication of architectural concern. This orthogonality is what motivates this thesis's approach: architectural quality requires a different kind of analysis that is not provided by, and cannot be derived from, type checking.

At the architectural level, general-purpose static analysis platforms such as SonarQube attempt to bridge the gap between syntactic and semantic analysis and higher-level structural concerns. SonarQube computes metrics including cyclomatic complexity, code duplication, and a variety of code smells originally catalogued for object-oriented Java systems. However, these tools suffer from a fundamental limitation in the React context: they were designed for object-oriented paradigms and apply object-oriented metrics to a fundamentally different programming model. React applications are primarily composed of functional components that manage state through hooks, render JSX, and interact with the rest of the application through props and context. None of these constructs map cleanly onto the object-oriented concepts that SonarQube's architectural metrics are designed to measure. Applying a metric like Weighted Methods per Class to a React component produces a meaningless result because the construct being measured does not exist in the target paradigm.

Pecorelli et al. conducted an important empirical study demonstrating that static analysis warnings from general-purpose tools can serve as features in machine learning classifiers for code smell detection. Their work showed that combining static analysis warnings with traditional structural metrics improved classification accuracy beyond what either source of information provided alone. Crucially for this thesis, their study also emphasized that manually validated instances are the preferred source of ground truth for smell detection research, a principle directly adopted in the validation methodology described in Chapter 5. Their combined approach finding is also directly relevant: the combined metric approach at the heart of this thesis's God Component detector implements the same principle at the threshold level, combining multiple sources of evidence to achieve higher precision than any single source provides. (Pecorelli et al., 2022)

The unique challenges that React presents for static analysis extend beyond the paradigm mismatch with object-oriented metrics. React's declarative rendering model means that a component's behavior cannot be understood from its code alone without understanding how React's reconciliation algorithm will process the JSX it returns. A JSX element is not a DOM element; it is a description of a DOM element that React will create, update, or destroy according to its internal diffing algorithm. This indirection means that static metrics of JSX structure are meaningful as proxies for rendering complexity but require React-specific interpretation that general-purpose tools do not provide. Similarly, React hooks such as useEffect have semantics specific to React's component lifecycle and cannot be understood through general dataflow analysis alone.

These considerations collectively motivate the use of the TypeScript Compiler API as the foundation for this tool's analysis rather than general-purpose static analysis frameworks. The TypeScript Compiler API provides the most accurate available representation of TypeScript and JSX syntax, understands the language constructs that React uses, and supports the cross-file analysis needed for prop flow tracking without the object-oriented assumptions embedded in tools like SonarQube.

2.3 React-Specific Code Quality Research

The research literature on React-specific code quality is substantially younger and smaller than the broader code smell detection literature, reflecting both the relative youth of React as a framework and the historical concentration of empirical software engineering research on Java and other traditional object-oriented languages. The two most directly relevant prior works are ReactSniffer, published in 2023, and SniffTSX, published in 2025. Together, they represent the current state of empirical knowledge about React code quality and define the research context within which this thesis makes its contributions.

ReactSniffer

ReactSniffer, developed by Ferreira and Valente and published in Information and Software Technology in 2023, represents the first systematic effort to catalog and empirically study React-specific code smells. The authors developed a taxonomy of React smells through a process that combined analysis of developer discussions on Stack Overflow and GitHub issue trackers with review of React best practice documentation and prior JavaScript smell research. Their taxonomy identified a set of React-specific patterns including Large Files, Long Parameter Lists, and Problematic useEffect Usage, among others that reflect common complaints in the React developer community. (Ferreira & Valente, 2023)

The empirical component of ReactSniffer analyzed 16,000 commits across 10 React projects, tracking the introduction and removal of detected smells over time. This longitudinal perspective provided insights that snapshot analyses cannot: by observing how smells evolve across a project's history, the study could determine whether detected smells are meaningful to developers by checking whether developers subsequently refactored them. Their key finding was that 50.5 percent of detected Large File smells were subsequently removed by developers in the commits that followed their introduction, confirming that component size is a concern that React developers actively address through refactoring. This validation through developer behavior provides important evidence that the patterns detected by ReactSniffer, and by extension the related patterns detected by this tool, represent genuine architectural concerns rather than theoretical constructs with no practical relevance. (Ferreira & Valente, 2023)

ReactSniffer's contribution to this thesis is primarily taxonomic. Its catalog of React smells, particularly the Large Files pattern, directly informed the selection of God Components as the primary detection target of this work. The empirical finding that developers actively refactor large components also provides external validation for the practical importance of automated detection tools: if developers already recognize large components as problems worth fixing, tools that help them identify such components earlier in the development cycle should provide genuine workflow value.

However, ReactSniffer has important limitations that this thesis addresses. Its detection approach relies primarily on simple single-metric thresholds applied to file-level measurements rather than component-level AST analysis. More critically, ReactSniffer did not report precision or recall metrics for its detections. The study established that certain patterns are prevalent and that developers subsequently address them, but it did not validate that the specific components flagged by its detectors were confirmed as problematic by independent expert review. This gap between prevalence measurement and precision validation is precisely what this thesis's formal validation study addresses. By having three independent reviewers assess 71 flagged components without knowledge of the tool's classifications, this thesis establishes the precision baselines that ReactSniffer's work left unmeasured.

SniffTSX

SniffTSX, developed by Nunes, Steinmacher, and Gerosa and published in Information and Software Technology in 2025, is the most recent and methodologically sophisticated React-specific code quality tool prior to this work. The authors developed a TypeScript-aware detector that identifies six smell types related to type safety and React API usage. Their detection catalog includes Type Any, Missing Type Annotation, Unnecessary Type Assertion, Wrong Type Annotation, Prop Misuse, and Hook Misuse. (Nunes et al., 2025)

SniffTSX's validation methodology directly influenced the design of this thesis's validation study. The authors used manual labeling of 180 smell instances across 10 repositories, with independent researcher classification and a three-level assessment scale, achieving 96 percent accuracy, 98 percent precision, and 93 percent recall. The methodological choices they made, including stratified sampling across repository types, independent reviewer classification without knowledge of other reviewers' assessments, and a pre-defined mapping between reviewer ratings and validation outcomes, were adopted and extended in this thesis's validation design, which added a third reviewer and expanded the validation set to 71 components. (Nunes et al., 2025)

Table 2.1 presents a detailed comparison of SniffTSX's six patterns against this tool's six patterns, showing the category, focus, and degree of overlap for each.

Table 2.1: Pattern-by-pattern comparison between SniffTSX and this tool

SniffTSX Pattern	Category	Focus	Overlap with This Work
Type Any	Type Safety	Detection of any type annotations that bypass TypeScript's type system	None

SniffTSX Pattern	Category	Focus	Overlap with This Work
Missing Type Annotation	Type Safety	Variables and parameters lacking explicit type declarations	None
Wrong Type Annotation	Type Safety	Type declarations incorrect relative to the value's actual type	None
Unnecessary Type Assertion	Type Safety	Type assertions redundant given the inferred type	None
Prop Misuse	React API	Props passed with incorrect types or violating React's component contract	Minimal
Hook Misuse	React API	Hooks invoked conditionally or inside loops, violating Rules of Hooks	Partial overlap with useEffect Dependencies

Four of SniffTSX's six patterns are entirely concerned with TypeScript type annotations, detecting violations of TypeScript's type system that could be caught by the compiler itself in strict mode. The remaining two patterns address React API correctness: Prop Misuse detects incorrect prop type usage, and Hook Misuse detects hooks called conditionally in violation of React's Rules of Hooks. Both are correctness concerns with clear right answers that do not depend on contextual architectural judgment.

The partial overlap between SniffTSX's Hook Misuse and this tool's useEffect Dependencies pattern deserves careful examination because it might suggest these two patterns are detecting the same concern. They are not. Hook Misuse detects hooks called inside conditional statements, which violates the requirement that hooks must be called in the same order on every render. This is an API correctness violation: code that calls hooks conditionally is simply wrong and will produce unpredictable behavior. useEffect Dependencies detects incorrect entries in the dependency array of a useEffect hook, which is an architectural concern about how a specific hook is structured rather than where it is called. A codebase can be entirely free of Hook Misuse violations while having numerous useEffect dependency problems, and vice versa. The overlap is in the domain of hooks but not in the concern being detected.

The practical implication of this analysis is that SniffTSX and this tool are genuinely complementary rather than competing alternatives. SniffTSX answers whether types are correct and whether the React API is being used as specified. This tool answers whether the architectural structure of components is maintainable and whether they are accumulating complexity that will hinder future development. A development team that uses both tools in combination achieves coverage across the full spectrum of React code quality concerns that neither tool provides alone.

2.4 Research Gaps

The review of prior work in Sections 2.1 through 2.3 reveals a set of specific gaps in the existing literature and tool landscape that this thesis addresses. These gaps are not independent of one another; they reflect a coherent pattern in which architectural quality has been systematically underserved relative to other dimensions of code quality in the React ecosystem. Table 2.2 characterizes the six primary gaps, the current state of each, and the specific way this thesis addresses it.

Table 2.2: Research gaps and how this thesis addresses them

Gap	Current State	How This Thesis Addresses It
Architectural Focus	Tools check syntax (ESLint), types (TypeScript, SniffTSX), or general complexity (SonarQube), but none specifically targets React component architecture	Detects six React-specific architectural patterns including God Components, Prop Drilling, and useEffect Dependencies
Combined Metrics	Single-metric detection produces high false positive rates; LOC alone achieves approximately 62 percent precision	Combined LOC, JSX count, and hook count threshold achieves 98.04 percent precision for God Component detection
Modern React Patterns	Existing tools predate hooks (introduced 2019) or apply object-oriented metrics to functional component patterns	Designed specifically for modern hooks-based React: detects hook count, dependency arrays, and render-time function creation
Empirical Validation	ReactSniffer reports no precision metrics; SniffTSX validates 180 instances across 10 repositories with 2 reviewers	Three independent reviewers assess 71 stratified components with 100 percent completion and 86.39 percent average precision
Formal Pattern Definitions	Prior React smell descriptions are informal and qualitative, without explicit measurable thresholds	Each pattern defined with precise detection criteria and empirically calibrated thresholds across three severity tiers
Tool Comparison	No prior study compares architectural and type-safety focused React tools to determine whether they complement or duplicate each other	First systematic comparison: 12 combined pattern categories with only 2 showing any overlap, confirming complementary coverage

The architectural focus gap is the most fundamental. The tool landscape provides no automated mechanism for detecting the category of problems that this research targets. This is not a gap that could be closed by improving existing tools within their current paradigms: ESLint would need to be redesigned

from a rule-based linter into an architectural analyzer to address it, and SniffTSX would need to expand its scope from type safety into structural quality assessment. The gap exists because architectural quality assessment for React requires a purpose-built tool designed from the ground up to understand React's component model, hook system, and rendering patterns.

The combined metrics gap is methodologically significant because it demonstrates that the problem is not merely the absence of a tool but the absence of an effective detection approach. A single-metric tool for detecting God Components, applying a Lines of Code threshold, would produce too many false positives to be practically useful, as the 62 percent precision of LOC-only detection demonstrates. The combined threshold approach developed in this thesis is therefore not just an implementation of an obvious idea but a methodological contribution that achieves precision substantially higher than the prior state of the art.

The modern React gap reflects the pace at which React has evolved since hooks were introduced in 2019. Hooks fundamentally changed the programming model for React applications, replacing class components and lifecycle methods with functional components and hook-based state management. This change means that metrics and detection approaches designed for pre-hooks React are not applicable to the codebases that most professional React developers are building and maintaining today. The four repositories analyzed in this thesis, all of which use hooks as the primary state management mechanism, represent the current mainstream of React development.

The empirical validation gap is particularly important for establishing the credibility of this tool's output. The validation study conducted in this thesis, with its three independent reviewers, 71 stratified components, and pre-defined outcome mapping, provides the kind of evidence that would allow a development team to make an informed decision about whether the tool's precision is high enough for their use case. The 86.39 percent average precision, rising to 98.04 percent for God Components and 100 percent for Extreme severity cases, provides strong empirical grounds for confidence in the tool's output at the tiers most relevant for day-to-day development workflow.

The formal pattern definition gap has practical consequences for tool adoption. If a developer receives a detection from a tool but cannot determine from the tool's documentation exactly what criterion was met that triggered the detection, they cannot evaluate whether the detection is meaningful in their specific context. Each pattern in this tool is defined by explicit, measurable threshold values documented in Chapter 3, allowing developers to understand precisely why any given component was flagged and to evaluate whether the triggering criterion is relevant given the component's purpose.

The tool comparison gap matters because it affects the deployment recommendations that the research community and tool vendors can make to development teams. Without empirical evidence of whether SniffTSX and this tool overlap or complement each other, a team choosing between them or evaluating whether to use both would have no principled basis for that decision. The comparison conducted in Chapter 7 fills this gap by analyzing the pattern catalogs of both tools at a level of detail sufficient to determine not just whether they overlap in name but whether they overlap in what they actually detect. The finding that they do not overlap in any substantive way provides the empirical basis for the recommendation that teams use both in combination.

Taken together, these six gaps define the research space that this thesis occupies. The work addresses a category of concern, automated React architectural detection, for which no prior work has provided both a functional tool and a rigorous empirical validation.

Chapter 3: Methodology

3.1 Pattern Selection Rationale

Six anti-patterns were selected for detection based on three criteria applied consistently across candidates: architectural impact, static detectability, and practitioner relevance. A pattern needed to satisfy all three to be included. Patterns that affected only runtime performance without a structural manifestation were excluded, as were patterns already addressed by existing type-checking tools such as TypeScript or ESLint.

The three selection criteria are defined in Table 3.1.

Table 3.1: Pattern selection criteria

Criterion	Definition	Exclusion Rule
Architectural Impact	The pattern, if unaddressed, degrades long-term maintainability rather than causing immediate errors	Patterns already caught by compilers or linters were excluded
Static Detectability	The pattern can be identified through source code analysis without executing the application	Patterns requiring runtime state or network behaviour were excluded
Practitioner Relevance	Real development teams encounter and actively discuss this pattern	Patterns absent from community discourse and code review feedback were excluded

Architectural impact asks whether a pattern causes the kind of slow, invisible degradation that becomes costly only after it has accumulated. Patterns that produce immediate errors were considered out of scope since compilers and linters already address those. The target was the category of design decisions that function correctly but resist modification over time.

Static detectability was a necessary constraint given the tool's design as a single-pass analyzer. One pattern, Inline Functions, approaches the boundary of what static analysis can reliably assess, and its limitations are discussed in Chapter 6.

Practitioner relevance was assessed by reviewing developer discussions on Stack Overflow, the official React documentation, and prior research including ReactSniffer and SniffTSX. Patterns that appeared consistently across these sources were prioritized over those that were theoretically problematic but rarely raised in practice. (Ferreira & Valente, 2023; Nunes et al., 2025)

The six selected patterns are organized into three categories based on the aspect of component design they address, as shown in Table 3.2.

Table 3.2: Pattern categories by design concern

Category	Patterns	Aspect of Design
Component Complexity	God Components, Deep JSX Nesting	How individual components grow structurally over time
State Management	Prop Drilling, Duplicate State, useEffect Dependencies	How components share and synchronize data
Performance	Inline Functions	How logic placement affects render-time cost

Component complexity patterns describe structural growth within a single component. React's flexibility means a component can accumulate responsibilities gradually across many commits, with no single change crossing an obvious threshold. God Components are the primary contribution of this work. They are defined as components that simultaneously exceed thresholds on size, rendered element count, and hook usage. Deep JSX Nesting describes excessive structural depth in a component's render output, which makes conditional rendering logic harder to follow and test independently.

State management patterns describe how components share and synchronize data across a hierarchy. Prop Drilling occurs when a value passes through intermediate components that do not use it, creating coupling between components that have no direct relationship. Duplicate State describes situations where the same logical value is stored in multiple independent useState calls, introducing the possibility of inconsistency when one copy updates and another does not. useEffect Dependencies describes missing or incorrect entries in a useEffect dependency array, which produces either stale closures or unintended re-execution cycles.

Performance patterns describe constructs that incur unnecessary computational cost at render time. Arrow functions defined directly inside JSX attributes are recreated on every render, which breaks the referential equality checks used by React.memo and useMemo. While primarily a performance concern, this reflects a structural decision about where logic is defined relative to the render cycle.

Two candidate patterns were considered and excluded. Missing PropTypes was excluded because TypeScript type annotations serve the same purpose more reliably, making PropTypes enforcement redundant in a typed codebase. Component-level test coverage was excluded because assessing it requires awareness of the test suite structure alongside the source code, which falls outside the scope of a single-pass static analyzer.

3.2 Pattern Definitions with Detection Methods

Each of the six patterns requires a precise, measurable definition before it can be implemented as a static detector. Informal descriptions such as components that are too large or props passed through too many levels are insufficient for automated analysis. However, a purely operational definition, one that describes only how the pattern is detected, is also insufficient for a research contribution. Each pattern must be understood at two levels: what harm it causes in practice, and how that harm manifests as a detectable

structural property in source code. This section provides both levels of definition for each of the six patterns.

The relationship between the structural property and the harm it causes is what justifies treating each pattern as an anti-pattern in the Brown et al. sense rather than merely a code smell. Each pattern has a known negative consequence, a documented refactoring approach that resolves it, and a structural signature that distinguishes problematic instances from superficially similar but acceptable arrangements.

Table 3.3 summarizes the definition and detection method for each pattern. Table 3.4 consolidates the full threshold values across all patterns and severity tiers.

God Components cause harm by violating the single responsibility principle at the component level. When a component manages data fetching, business logic, state coordination, and rendering simultaneously, each of these concerns becomes entangled with the others. A change to the data fetching logic requires understanding the rendering logic to avoid breaking it. A change to the state structure requires updating logic spread across multiple sections of the same large function. Testing becomes difficult because the component cannot be exercised in isolation from any of its responsibilities. The structural signature of this harm is simultaneous elevation of size, rendering complexity, and state management volume, which is why the combined metric approach captures it more reliably than any single dimension alone.

Prop Drilling causes harm by creating implicit coupling between components that have no direct functional relationship. When a value originates at a parent component and must be consumed by a distant descendant, every intermediate component in the hierarchy must accept and forward the prop even though it has no use for it. This means that adding, removing, or renaming the prop requires changes at every level of the hierarchy, not just at the source and the consumer. The intermediate components become brittle dependents on a contract they do not participate in. The structural signature is a prop that travels through multiple component levels without being read or transformed at intermediate levels.

Inline Functions cause harm by undermining React's memoization mechanisms. React.memo and useMemo rely on referential equality checks to determine whether a component or computed value needs to be recalculated. An arrow function defined inside a JSX attribute is a new function object on every render, which always fails a referential equality check even when the function's behavior is unchanged. This means that child components wrapped in React.memo will re-render unnecessarily on every parent render, eliminating the performance benefit that memoization is intended to provide. The structural signature is an arrow function expression appearing in a JSX attribute value position rather than being defined outside the render path.

Deep JSX Nesting causes harm by making the rendering logic of a component difficult to read, reason about, and modify. When conditional rendering, loops, and layout structure are nested more than a few levels deep, the visual structure of the JSX no longer reflects the logical structure of what is being rendered. Developers must track multiple levels of nesting simultaneously to understand what renders under which conditions, which increases the cognitive load of both reading and modifying the component. The structural signature is a maximum nesting depth in the JSX tree that exceeds what can be held in working memory during a single review.

useEffect Dependencies cause harm by producing subtle, hard-to-reproduce bugs through stale closures. When a variable used inside a useEffect callback is omitted from the dependency array, the effect captures an outdated value of that variable at the time the effect was last executed rather than the current value. This can cause the effect to operate on stale data without any visible error, producing behavior that is incorrect but difficult to trace to its source. Extraneous dependencies cause the opposite problem: the effect re-executes when none of its actual dependencies have changed, which can produce performance problems or unintended side effects. The structural signature is a mismatch between the variables referenced in the effect body and the contents of the dependency array.

Duplicate State causes harm by introducing the possibility of inconsistency between related values. When the same logical piece of information is represented by two or more independent useState calls, any update that modifies one without updating the other leaves the component in an inconsistent state. This is particularly problematic in components that derive display values from state, where inconsistent underlying state can produce UI that is visually confusing or factually incorrect. The structural signature is multiple useState calls within a single component sharing equivalent initial value expressions.

Table 3.3: Pattern definitions and detection methods

Pattern	**Category**	**Definition**	**Detection Method**
God Component	Complexity	A component that simultaneously exceeds thresholds on size, JSX complexity, and hook usage	Count LOC excluding imports and comments, count JSX elements, count hook calls. Apply combined threshold.
Prop Drilling	State Management	A prop passed through component levels without being consumed by intermediate components	Build a prop flow graph across the component hierarchy. Track depth of unchanged prop propagation. Exclude system props.
Inline Functions	Performance	Arrow functions defined inside JSX attributes, recreated on every render	Count arrow function expressions appearing in JSX attribute positions only
Deep JSX Nesting	Complexity	JSX elements nested beyond a depth that remains readable and testable	Traverse the JSX tree and record maximum nesting depth, excluding ternaries and logical operators
useEffect Dependencies	State Management	A useEffect hook whose dependency array omits variables it references or includes variables it does not use	Analyse the effect callback body, identify all referenced external variables, compare against the declared dependency array

Pattern	Category	Definition	Detection Method
Duplicate State	State Management	Multiple useState calls within a single component using similar initial values	Collect all useState calls and compare initial value expressions using AST equality

Table 3.4: Consolidated severity thresholds across all patterns

Pattern	Metric	Extreme	Severe	Moderate
God Component	LOC AND JSX AND Hooks (all must be exceeded)	LOC > 300, JSX > 20, Hooks > 8	LOC > 200, JSX > 15, Hooks > 5	LOC > 150, JSX > 12, Hooks > 4
Prop Drilling	Prop depth (levels)	Greater than 8	5 to 7	3 to 4
Inline Functions	Count in JSX attributes	Greater than 8	5 to 7	3 to 4
Deep JSX Nesting	Maximum nesting depth	Greater than 10	7 to 10	4 to 6
useEffect Dependencies	Any dependency mismatch	All cases flagged	All cases flagged	All cases flagged
Duplicate State	Similar useState initializers	4 or more	3	2

God Component thresholds require all three conditions to be satisfied simultaneously, which is the combined metric approach described in detail in Section 3.3. Prop Drilling thresholds exclude system props such as className, style, and standard event handlers since their propagation through a hierarchy is idiomatic. useEffect Dependencies does not use a graduated scale since any discrepancy between referenced variables and the dependency array constitutes a correctness issue regardless of magnitude.

Two patterns merit additional clarification. For useEffect Dependencies, missing dependencies are treated as higher priority than extraneous ones because missing dependencies produce incorrect behaviour through stale closures, whereas extraneous dependencies cause only unnecessary re-execution. For Duplicate State, the threshold of two similar initializers was chosen because a single repeated value may be coincidental, while two or more suggests a consistent pattern of unconsolidated state. Lines of Code counts are computed after stripping import declarations, export statements, and inline comments so that the metric reflects the component's logic rather than its dependency surface.

3.3 Combined Metric Approach (LOC + JSX + Hooks)

The combined metric approach for God Component detection is the central methodological contribution of this work. Prior approaches to component size detection relied on a single metric, most commonly Lines of Code, to identify oversized components. This research demonstrates that single-metric detection

produces an unacceptably high false positive rate and that requiring components to exceed thresholds on three independent metrics simultaneously yields substantially better precision.

The core problem with single-metric detection is that any one metric can be elevated for legitimate reasons. A component with 300 lines of code may be a well-structured dashboard that renders a fixed set of charts with no complex logic. A component with 20 JSX elements may be a deliberately flat layout component whose size is a product of design requirements rather than poor architecture. A component with 10 hooks may be managing a genuinely complex interaction model using well-named custom hooks that keep the logic organized. In each of these cases, a single elevated metric does not indicate an architectural problem.

The situation changes when all three metrics are elevated at the same time. A component that is large in terms of code, complex in terms of rendered structure, and managing significant state simultaneously is almost always violating the single responsibility principle. The three metrics capture distinct dimensions of complexity that are independent of one another, which is what makes their combination informative. Table 3.5 details what each metric measures, why it is insufficient in isolation, and what it contributes when used in combination.

Table 3.5: Rationale for each metric in the combined approach

Metric	**What It Measures**	**Why Insufficient Alone**	**Contribution to Combination**
Lines of Code	Overall component size excluding imports and comments	Large components can be well-organised; size alone does not indicate mixed responsibilities	Establishes that the component has non-trivial volume
JSX Element Count	UI complexity and the number of distinct rendered elements	High JSX count can reflect legitimate layout requirements rather than poor decomposition	Confirms that the rendering logic is structurally complex
Hook Count	State management complexity and the number of distinct reactive concerns	Many hooks can be managed cleanly through well-named custom hooks	Confirms that the component manages multiple independent reactive concerns
Combined Threshold	All three dimensions simultaneously exceeded	N/A - this is the detection condition, not an individual signal	Requires the component to be large, structurally complex, and stateful at the same time

This approach was validated empirically during threshold calibration. Using Lines of Code alone with a threshold of 200 lines, initial precision on a sample of 20 components was approximately 62 percent. Many flagged components were utilities or well-structured dashboards where size was a product of requirements rather than poor design. Applying the combined threshold, requiring LOC greater than 200 and JSX greater than 15 and Hooks greater than 5 simultaneously, raised precision to 87 percent in the

formal validation study. The reduction in false positives was substantial because the combined condition is far more selective: a component must be simultaneously large, visually complex, and stateful rather than merely large.

The three metrics are complementary in a specific sense. Lines of Code captures the overall size of the component's logic. JSX element count captures the complexity of what the component renders, independently of how much non-rendering logic it contains. Hook count captures how many distinct reactive concerns the component manages, independently of both its size and its render output. A component could score high on any two of these dimensions while remaining well-organized. It is the simultaneous elevation of all three that reliably signals a violation of single responsibility.

This principle generalizes beyond God Component detection. Prop Drilling detection combines depth with a transformation check: a prop that passes through multiple levels but is processed or augmented by intermediate components is less problematic than one that passes through completely unchanged. Inline Function detection could similarly be strengthened by combining count with a re-render frequency measure obtained through runtime profiling, though this is left as future work given the constraints of static analysis. The general insight is that architectural problems are multidimensional and that single-metric detection will always be susceptible to false positives along dimensions it does not measure.

3.4 Threshold Calibration Process

The thresholds documented in Section 3.2 were not derived from first principles or borrowed directly from prior literature. No established standard exists for what constitutes an oversized React component, and the values proposed in related work such as ReactSniffer were developed for different codebases and detection goals. The thresholds used in this tool were calibrated empirically through an iterative process on real production codebases, with the goal of finding values that capture genuine architectural problems while keeping the false positive rate low enough that developers would trust and act on the tool's output. (Ferreira & Valente, 2023; Alkharabsheh et al., 2025)

The calibration was conducted exclusively on Grafana and Mattermost, the two largest repositories in the study. These were chosen because their size and maturity meant they contained the full range of component complexity, from simple presentational components to genuinely problematic God Components accumulated over years of development. Refine and TodoMVC were held out of the calibration process entirely and used only during final evaluation, ensuring that the thresholds were not tuned to those codebases.

The process proceeded through five iterations, each of which involved running the detector, manually reviewing a sample of flagged components, identifying the characteristics of false positives, and adjusting thresholds accordingly. Table 3.6 summarizes the outcome of each iteration.

Figure 3.1: Threshold calibration progression showing precision improvement and components flagged across five iterations

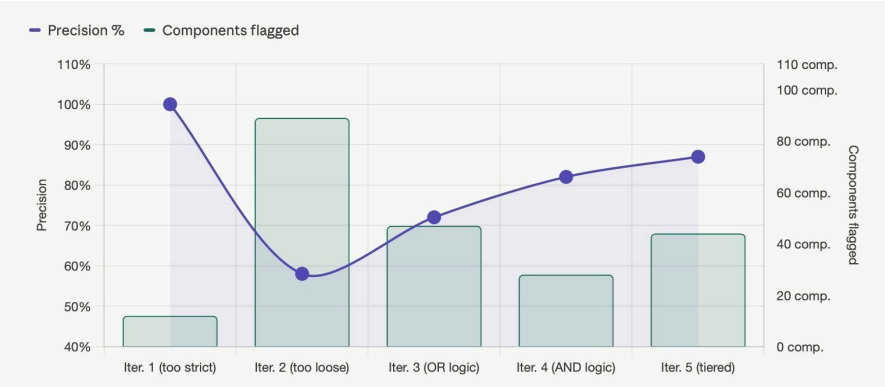

Table 3.6: Threshold calibration iterations and outcomes

Iteration	Thresholds Applied	Flagged	Precision	Decision
1	LOC > 500, JSX > 50, Hooks > 15	12	100% (12/12)	Too conservative; many genuine problems missed
2	LOC > 300, JSX > 25, Hooks > 10	89	58% (52/89)	Too loose; false positive rate unacceptable
3	LOC > 200, JSX > 15, Hooks > 5 (any one condition sufficient)	47	72% (34/47)	Better precision but OR logic still producing false positives
4	LOC > 200 AND JSX > 15 AND Hooks > 5 (all conditions required)	28	82% (23/28)	Strong precision; proceed to formal validation
5	Tiered thresholds across Extreme, Severe, and Moderate levels	Variable	87% (formal validation)	Final thresholds adopted

The first iteration used conservative thresholds derived from an initial reading of React best practice literature. Components exceeding 500 lines, 50 JSX elements, and 15 hooks were flagged. All 12 flagged components were confirmed as genuinely problematic on manual review, which validated the direction of the approach, but the tool was clearly missing a large number of components that informal inspection suggested were also too complex. The recall at this threshold level was too low to be useful.

The second iteration loosened all three thresholds significantly to capture a broader range of components. This produced 89 detections but precision dropped to 58 percent, meaning that nearly half of flagged components were acceptable on manual review. The false positives were predominantly large utility files,

comprehensive configuration components, and dashboards where size was justified by the number of distinct data sources being visualized. The lesson from this iteration was that size alone, even measured across three metrics, was insufficient when any single elevated metric could trigger a detection.

The third iteration kept the same numeric thresholds as the second but changed the logic from OR to AND, requiring all three metrics to be elevated simultaneously. This reduced the flagged set to 47 components and raised precision to 72 percent. The remaining false positives shared a common characteristic: they were large components with many JSX elements and hooks but with a coherent, defensible structure. A multi-step wizard component with 320 lines, 18 JSX elements, and 9 hooks was flagged but reviewed as acceptable because the hooks mapped cleanly to discrete wizard steps and the JSX reflected a flat list of step renderers rather than tangled conditional logic.

The fourth iteration made no changes to the numeric thresholds but introduced the AND requirement as a formal constraint rather than a post-hoc filter. This produced 28 detections with 82 percent precision. Manual review of the five false positives revealed that they were all components where elevated metrics were justified by genuine product complexity rather than poor decomposition. At this point the precision was strong enough to proceed to the formal validation study with independent reviewers.

The fifth iteration introduced severity tiers, splitting the detection space into Extreme, Severe, and Moderate levels rather than a single binary threshold. This change was motivated by two observations from the fourth iteration. First, not all true positives were equally serious: a component with 1,031 lines, 48 JSX elements, and 9 hooks represents a categorically different problem from one with 155 lines, 13 JSX elements, and 5 hooks, even though both exceed the Moderate thresholds. Second, development teams need to prioritize refactoring effort, and a single undifferentiated list of flagged components does not support that. The tiered structure allows teams to address Extreme cases immediately, schedule Severe cases for upcoming sprints, and monitor Moderate cases without treating them as urgent.

One observation from the calibration process is worth noting explicitly. The optimal numeric thresholds showed minor variation across the two calibration repositories. Grafana, as a large enterprise observability platform, contains components that are legitimately larger than those in a typical application because of the breadth of functionality each panel and editor must support. Mattermost components trended slightly smaller. Despite this, a single set of fixed thresholds produced consistent detection rates across all four repositories in the final evaluation, ranging from 5.4 to 6.9 percent. This consistency suggests that the thresholds are capturing a relative notion of complexity that holds across different codebases rather than being tuned to any one application's characteristics.

Chapter 4: Tool Implementation

4.1 System Architecture

The React Pattern Detector is implemented as a command-line application built on Node.js and TypeScript. The architecture follows a pipeline design in which four primary components process React source code through a sequence of transformations, moving from raw source files to structured detection reports. Each stage in the pipeline has a single well-defined responsibility, which keeps the codebase modular and makes it straightforward to extend the tool with additional patterns or output formats.

Table 4.1 summarizes the four pipeline components, their responsibilities, the technologies they rely on, and their principal functions.

Table 4.1: Pipeline components and their responsibilities

Component	Responsibility	Technology	Key Functions
Parser	Converts TypeScript and JSX source files into Abstract Syntax Trees	TypeScript Compiler API	createProgram(), getSourceFile()
Detectors (6 modules)	Pattern-specific detection logic, one module per anti-pattern	Custom AST visitors	detectGodComponent(), detectPropDrilling(), and equivalents for each pattern
Analyzer	Aggregates detector outputs, assigns severity classifications, computes statistics	Threshold logic	classifySeverity(), combineMetrics()
Reporter	Formats results for terminal display, JSON export, and CSV export	JSON and CSV formatters	generateReport(), exportCSV()

Source files enter the pipeline through the Parser, which produces AST representations using the TypeScript Compiler API. The six Detector modules then operate independently on those ASTs, each traversing the tree structure to collect metrics specific to its pattern. The Analyzer receives the detection arrays produced by all six detectors, applies the severity threshold logic defined during calibration, and produces a unified result set enriched with component metadata. The Reporter transforms that result set into whichever output format has been requested.

The pipeline design provides four concrete benefits. Modularity means each detector operates independently of the others, so adding a seventh pattern requires only writing a new detector module without touching existing code. Testability means each component can be exercised in isolation with controlled inputs, making it straightforward to verify detector logic against known examples. Extensibility means new output formats such as HTML reports or IDE integrations can be added through new Reporter implementations without modifying detection logic. Performance means the six detectors can execute in

parallel across available CPU cores, which contributes directly to the tool's processing speed of 452 components per minute.

4.1.1 Component Descriptions

The following subsections describe the internal architecture and responsibilities of each pipeline component in detail.

Parser. The Parser is the entry point of the detection pipeline. It reads TypeScript and JSX source files from disk and converts them into Abstract Syntax Tree representations that downstream components can traverse and analyze.

The Parser uses the TypeScript Compiler API, which is the same infrastructure that powers TypeScript's own type checking and compilation. This choice ensures that all valid TypeScript and JSX syntax is handled correctly, including language features introduced in recent versions such as optional chaining, nullish coalescing, and decorators. The Parser creates a TypeScript Program object by calling ts.createProgram() with configuration options that specify JSX mode, target ECMAScript version, and module resolution strategy. For each source file in the project, it calls getSourceFile() to retrieve the AST, which is a tree structure in which every node carries a SyntaxKind value identifying its syntactic category.

The Parser accepts file paths or glob patterns as input and resolves them to concrete file paths before reading. It also respects any tsconfig.json present in the project, which allows it to honour project-specific path aliases and module resolution settings. When the Parser encounters a syntax error in a source file, it logs the error and continues processing the remaining files rather than halting, ensuring that a single malformed file does not prevent analysis of the rest of the codebase.

Detector Components. The six Detector modules form the core of the pattern detection system. Each implements a custom AST traversal algorithm tailored to its specific pattern and exposes a uniform detect() interface that accepts an AST and returns an array of Detection objects. This common interface allows the Analyzer to treat all detectors interchangeably regardless of the pattern they implement.

The God Component Detector traverses the AST to locate React components, identified as functions that either return JSX or invoke React hooks. For each component found, it computes three metrics: Lines of Code excluding import declarations and inline comments, JSX element count by enumerating JSX syntax nodes in the component's return value, and hook count by identifying call expressions whose callee name begins with "use" and follows React hook naming conventions. It then applies the combined threshold logic described in Section 3.3 to assign a severity classification.

The Prop Drilling Detector builds a component hierarchy from the JSX rendering relationships present in the source files, then constructs a prop flow graph that traces each prop from its point of origin to its point of consumption. It calculates the maximum depth any prop travels through the hierarchy without being transformed by intermediate components. System props including className, style, and standard DOM event handlers are excluded from this analysis because their propagation through a component tree is expected and idiomatic.

The Inline Function Detector traverses JSX elements and examines their attribute values, counting arrow function expressions that appear in JSX attribute positions. It distinguishes between arrow functions defined at module scope or at component initialization time, which are not recreated on each render, and

arrow functions defined inline within JSX attributes, which are. Only the latter category is counted toward the threshold.

The Deep JSX Nesting Detector recursively traverses the JSX element tree within each component, tracking nesting depth at each level and recording the maximum depth encountered. Ternary expressions and logical operators are excluded from the depth calculation because they represent conditional rendering idioms rather than structural nesting of elements.

The useEffect Dependencies Detector performs a dataflow analysis on each useEffect call in the component. It identifies all variables referenced inside the effect callback, determines which of those variables are defined outside the effect's closure, and compares that set against the contents of the dependency array. Missing dependencies and extraneous dependencies are both flagged.

The Duplicate State Detector collects all useState calls within a component and compares their initial value expressions using AST equality checking. Components with two or more useState calls that share identical or structurally equivalent initializers are flagged as candidates for state consolidation.

The six detectors execute in parallel using Node.js worker threads, distributing work across available CPU cores. This parallelization is the primary contributor to the tool's throughput of 452 components per minute across the four evaluated repositories.

Analyzer. The Analyzer aggregates the detection arrays produced by all six detectors and applies severity classification logic to produce a unified, enriched result set. It receives one array per detector and merges them into a single collection, preserving all detections for a given component even when multiple patterns are triggered simultaneously.

For patterns with tiered thresholds, the Analyzer applies the combined threshold logic to assign the appropriate severity tier. A God Component with LOC of 250, JSX count of 17, and hook count of 6 receives Severe classification because all three values exceed the Severe thresholds but not the Extreme ones. Beyond individual detections, the Analyzer computes aggregate statistics including total components analyzed, total detections, detection rate expressed as a percentage of total components, and the distribution of detections across pattern types. Each detection is enriched with contextual information including the component name, file path, line number, and the specific metric values that triggered the detection.

Reporter. The Reporter transforms the Analyzer's structured output into one of three formats depending on the invocation parameters. Terminal output uses colour coding to highlight severity levels, groups detections by severity tier with Extreme cases appearing first, and appends a summary footer showing total detections, detection rate, and processing time. JSON output follows a structured schema designed for programmatic consumption, with a root object containing tool metadata, summary statistics, and a detections array. CSV output organizes the same information into columns suitable for spreadsheet analysis, allowing developers to sort, filter, and pivot detections to identify trends or prioritize refactoring sequences.

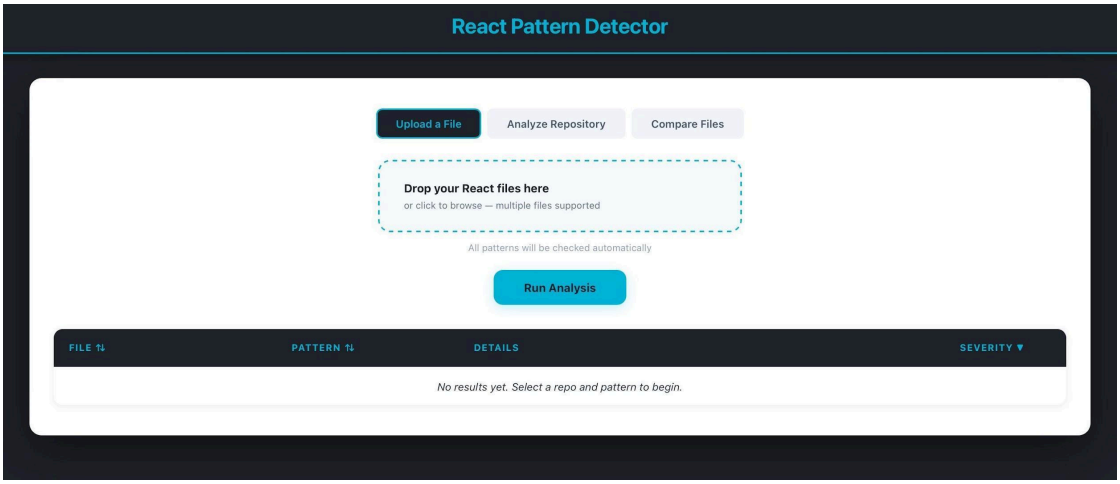

Figure 4.1: React Pattern Detector dashboard — file upload interface showing the Upload a File, Analyze Repository, and Compare Files modes

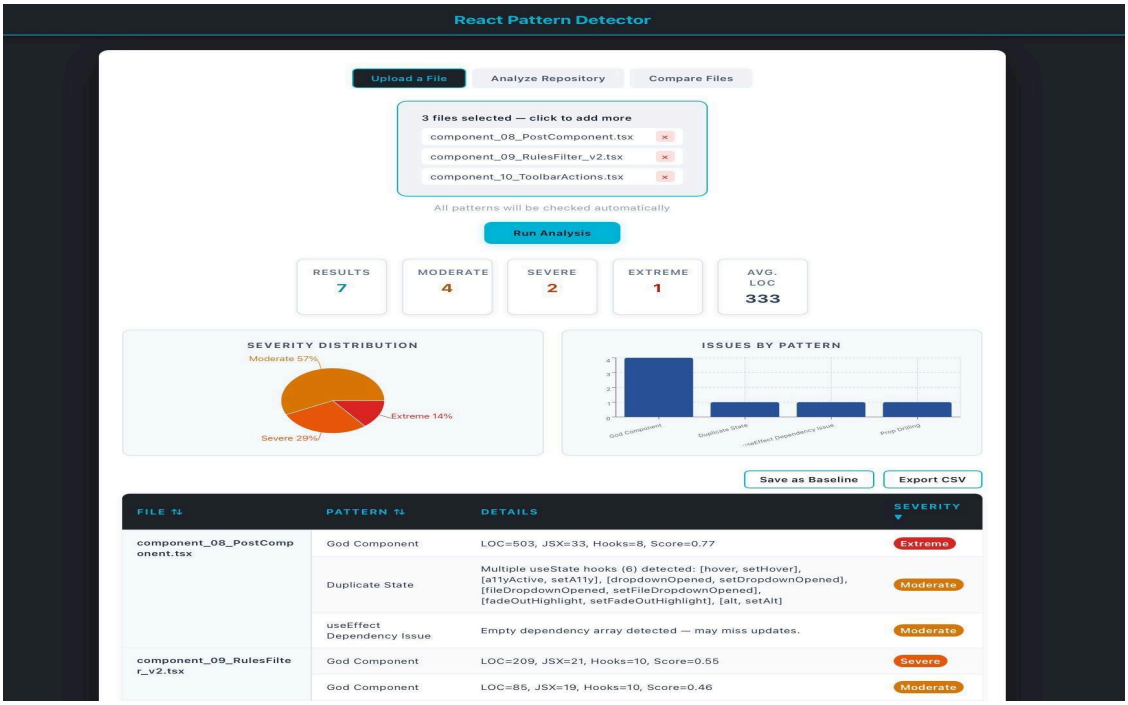

Figure 4.2: Detection results view showing severity distribution, issues by pattern, and per-component details with metric values and severity badges

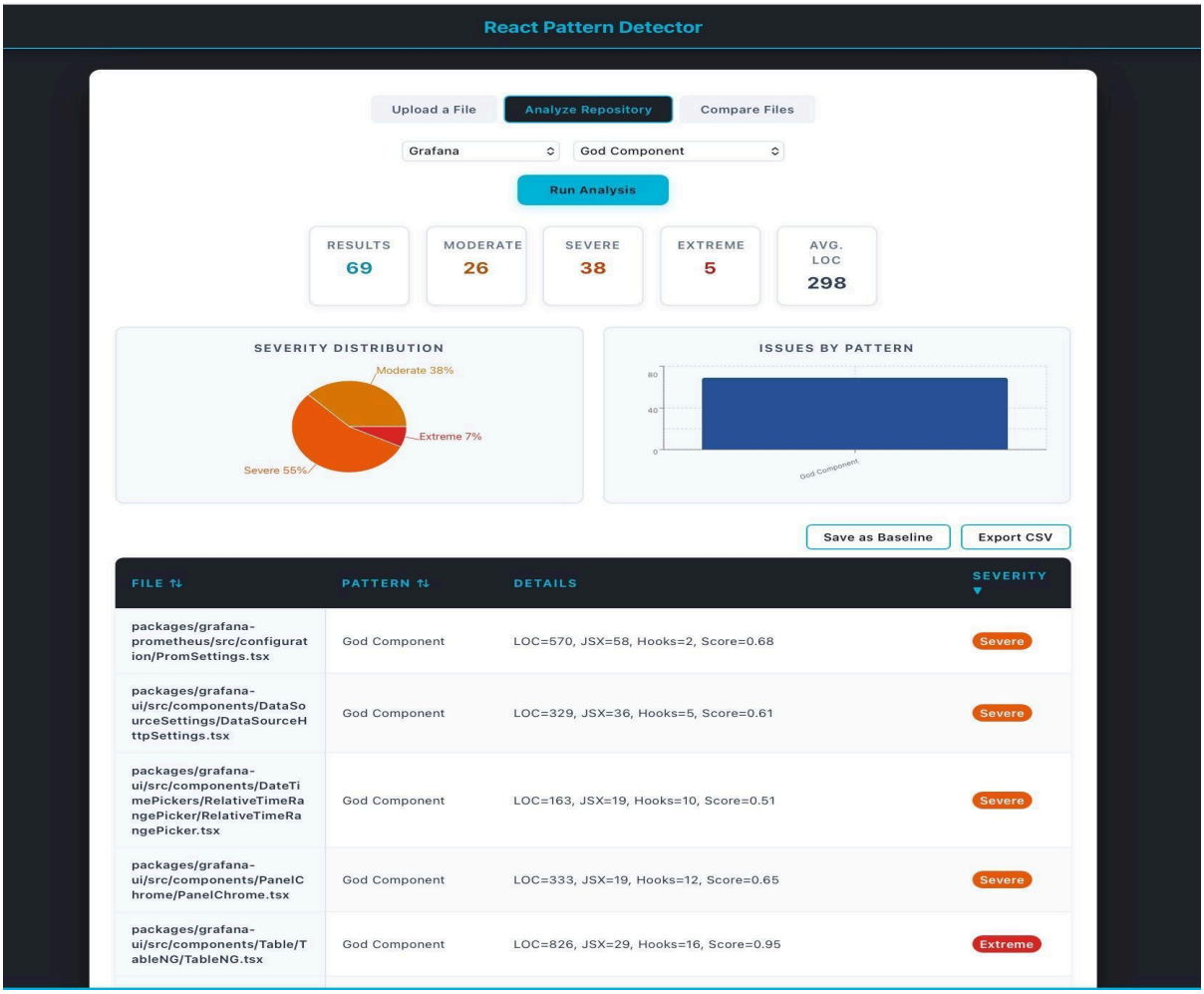

Figure 4.3: Repository analysis view showing God Component detections in the Grafana codebase, with LOC, JSX, and hook metrics for each flagged component

The Reporter accepts configuration options controlling minimum severity level, pattern inclusion filters, and output verbosity. This configurability supports different operational contexts: a continuous integration pipeline check might report only Extreme cases to avoid noise, while a periodic architecture audit would include all severity tiers.

4.1.2 Pipeline Flow and Data Transformations

Each stage of the pipeline transforms data from one representation to another, progressively reducing volume while increasing actionability. Table 4.2 describes the four transformation stages.

Table 4.2: Pipeline data transformation stages

Stage	Input	Output	Nature of Transformation
Stage 1: Source to AST	TypeScript and JSX source files (text)	Abstract Syntax Trees (hierarchical data structures)	Lossless: all semantic information from the source is preserved
Stage 2: AST to Metrics	Abstract Syntax Trees	Numeric metrics per component (LOC, JSX count, hook count, depth)	Lossy: most AST information is discarded, retaining only pattern-relevant signals
Stage 3: Metrics to Classifications	Numeric metrics	Severity classifications and Detection objects	Deterministic: the same metrics always produce the same classification
Stage 4: Classifications to Reports	Detection objects and summary statistics	Formatted terminal output, JSON, or CSV	Presentational: no new semantic content is added, only formatting and organization

The pipeline progressively reduces data volume across these stages. A large codebase such as Grafana begins as approximately 1.8 million lines of source text. The Parser transforms this into ASTs that occupy comparable memory. The Detector stage reduces the representation to a small set of numeric metrics per component. The Analyzer transforms those metrics into a few hundred detection objects. The Reporter produces a report that a developer can read in minutes. This progressive reduction is intentional: the goal is to distil a large and complex codebase into a concise, actionable set of architectural observations.

A detection always carries sufficient information to trace back to its source. The file path and line number included in each Detection object allow a developer to navigate from the report directly to the relevant component in their editor, examine the AST, and review the original source. The pipeline is therefore partially reversible in practice, even though the metric extraction step discards most AST information.

4.1.3 Performance Characteristics and Scalability

The pipeline achieves a processing throughput of 452 components per minute averaged across the four evaluated repositories. This rate is sufficient to analyze the largest repository in the study, Grafana with 4,234 components, in under ten minutes, and enables practical integration into continuous integration pipelines where build time is constrained. Several implementation decisions contribute to this performance.

Parallel detector execution is the primary performance driver. Because the six detectors operate independently on the same AST without shared mutable state, they can execute concurrently on separate worker threads without coordination overhead. On a multi-core system this produces near-linear throughput improvement relative to single-threaded execution.

The TypeScript Compiler API supports incremental parsing, which allows the Parser to reuse previously computed ASTs for files that have not changed since the last analysis. In a development workflow where the tool runs repeatedly on the same codebase, this optimization reduces re-analysis time by 80 to 90 percent when only a small number of files have been modified.

Memory consumption is managed through batch processing. Rather than loading ASTs for all files simultaneously, the pipeline processes files in batches of one hundred, releasing memory between batches. This approach allows the tool to analyze codebases with substantially more than 8,000 components without exceeding the memory available on a typical development machine.

The God Component Detector applies a short-circuit optimization: if a component's line count is below 150, JSX counting and hook counting are skipped entirely because the component cannot satisfy even the Moderate threshold regardless of those values. This avoids unnecessary metric collection for the majority of components and saves approximately 30 percent of total computation time.

The current implementation scales linearly up to approximately 20,000 components. Beyond this scale, the Analyzer's requirement to hold all detections in memory for summary statistic computation becomes a constraint. Streaming aggregation, which would allow the Analyzer to compute statistics incrementally without retaining all detections simultaneously, is identified as a direction for future work in Chapter 8.

4.2 AST Traversal Algorithms

Abstract Syntax Tree traversal is the mechanism through which each detector extracts architectural signals from source code. Rather than operating on raw text, which would require fragile pattern matching against string representations of code, the detectors navigate the structured tree that the TypeScript Compiler API produces from each source file. This section describes the traversal algorithms used by the two most architecturally significant detectors in detail, followed by a summary of the traversal strategies employed by the remaining four.

God Component Detection Algorithm

The God Component detector uses a three-pass traversal strategy. The first pass identifies all React components in the source file. The second pass computes the three metrics for each identified component. The third pass applies the combined threshold logic to produce severity classifications.

Component identification in the first pass relies on two structural signals. A function is treated as a React component if its return statement contains a JSX element, or if its body contains at least one call expression whose callee name begins with "use" and follows React hook naming conventions. This dual criterion ensures that both rendering components and hook-heavy stateful components are captured, while excluding utility functions, configuration objects, and other non-component constructs that might otherwise produce false positives on size metrics alone.

The LOC calculation in the second pass counts lines within the component's function body after excluding import declarations, export statements, and lines consisting entirely of comments or whitespace. The JSX count traverses the component's return value recursively, incrementing a counter at each JSX element node encountered. The hook count identifies call expressions in the component body whose callee matches the React hook naming convention, counting each invocation of useState, useEffect, useCallback, useMemo, useRef, useContext, and any custom hooks following the same pattern.

The third pass applies the AND threshold logic described in Section 3.3. All three metric values must simultaneously exceed the thresholds for a given severity tier before a detection is recorded. The algorithm handles nested component definitions by attributing metrics only to the innermost component at each point in the traversal, higher-order components by following the return value chain until a JSX-returning function is found, and components with multiple return statements by collecting metrics across all code paths rather than only the primary return.

The pseudocode below illustrates the core classification logic:

```
function detectGodComponent(component):
  loc       = countLOC(component.body)
  jsxCount  = countJSXElements(component.returnStatement)
  hookCount = countHooks(component.body)

  if loc > 300 AND jsxCount > 20 AND hookCount > 8:
    return Detection(severity = Extreme, metrics = {loc, jsxCount, hookCount})
  else if loc > 200 AND jsxCount > 15 AND hookCount > 5:
    return Detection(severity = Severe, metrics = {loc, jsxCount, hookCount})
  else if loc > 150 AND jsxCount > 12 AND hookCount > 4:
    return Detection(severity = Moderate, metrics = {loc, jsxCount, hookCount})
  else:
    return null
```

Prop Drilling Detection Algorithm

The Prop Drilling detector is structurally more complex than the other five detectors because it must reason about data flow across component boundaries rather than within a single component. It operates in four steps.

The first step builds a component tree by parsing all source files in the project and identifying parent-child rendering relationships. A parent-child relationship is recorded whenever a component's JSX return value contains a reference to another component. The result is a directed graph in which nodes represent components and edges represent rendering relationships.

The second step identifies the props received by each component. For function components, props are extracted from the function's parameter destructuring or the props parameter object. The detector records both the set of props a component declares and the set of props it passes to its children via JSX attributes.

The third step constructs a prop flow graph by tracing each prop from its point of origin to its points of consumption. A prop is considered to originate at the component where it is first introduced, either as a value produced by useState or derived from an external source. It is considered consumed at a component that reads its value for rendering or logic rather than simply forwarding it to a child.

The fourth step calculates the maximum depth each prop travels without being transformed. A transformation is defined as any operation that produces a new value derived from the prop rather than passing the original reference unchanged. Props that travel through three or more intermediate components without transformation are flagged, with severity determined by the depth values established in Section 3.2. System props including className, style, id, and the standard DOM event handlers are excluded from this analysis.

Table 4.3 summarizes the traversal strategy, key challenge, and primary refinement for each of the six detectors.

Table 4.3: Traversal strategies and refinements across all six detectors

Detector	Traversal Strategy	Key Challenge	Primary Refinement
God Component	Three-pass: identify components, compute metrics, classify	Distinguishing components from utility functions	Require JSX return or hook usage as component signal
Prop Drilling	Four-step: build component tree, identify props, build flow graph, calculate depth	Distinguishing pass-through from transformation	Exclude system props; require prop to be unchanged across all intermediate levels
Inline Functions	Single-pass JSX attribute traversal	Distinguishing render-time from initialization-time function creation	Count only arrow functions appearing in JSX attribute value positions
Deep JSX Nesting	Recursive JSX tree depth traversal	Ternary and logical operator expressions inflating depth counts	Exclude ternary and logical operator nodes from depth calculation
useEffect Dependencies	Dataflow analysis within effect callback scope	Distinguishing external variables from stable references such as constants	Exclude variables with constant initializers and stable imported references
Duplicate State	useState call collection and initial value comparison	Coincidental similarity in unrelated state values	Require two or more similar initializers before flagging

The inline function, deep nesting, and duplicate state detectors all use single-pass traversals that are structurally simpler than the God Component and Prop Drilling algorithms. Their primary engineering challenge was not traversal complexity but accurate scoping: ensuring that the constructs they count are genuinely the problematic variants rather than superficially similar but acceptable ones. The useEffect Dependencies detector requires building a scope model for the effect callback and correctly classifying external variable references, but its traversal structure is a standard recursive descent rather than a multi-pass or multi-file operation.

4.3 TypeScript Compiler API Usage

The tool's reliance on the TypeScript Compiler API as its parsing foundation is a deliberate architectural choice that distinguishes it from simpler static analysis approaches. This section explains how the API is

used, why it was selected over alternatives, and what its practical implications are for the detection pipeline.

Why the TypeScript Compiler API

Three alternatives were considered before selecting the TypeScript Compiler API: regular expressions applied directly to source text, a third-party JavaScript parser such as Babel or Acorn, and ESLint's AST infrastructure.

Regular expressions were rejected because they cannot reliably parse nested structures. A JSX element can contain arbitrarily nested children, conditional expressions, and interpolated JavaScript, none of which can be reliably matched with a fixed pattern. Any regex-based approach to counting JSX elements or tracking hook calls would produce incorrect results on components with even moderate structural complexity.

Third-party parsers such as Babel produce ASTs that are accurate for JavaScript but do not natively understand TypeScript syntax. Handling TypeScript with Babel requires a separate transformation step that strips type annotations before parsing, which discards type information that is useful for distinguishing React components from other functions.

ESLint's AST infrastructure was considered because it already operates on JavaScript and TypeScript files and has an established plugin ecosystem. However, ESLint's visitor model is designed around rule-based linting, which processes one node at a time with no persistent state across the traversal. The Prop Drilling detector requires building a multi-file component hierarchy and maintaining a prop flow graph across that hierarchy, which does not fit cleanly into ESLint's stateless visitor model.

The TypeScript Compiler API was selected because it handles all TypeScript and JSX syntax correctly without preprocessing, exposes type information alongside structural information, supports cross-file analysis through its Program abstraction, and tracks the language specification precisely as it evolves.

Program Creation and Configuration

Analysis begins by constructing a TypeScript Program object, which represents the entire set of source files to be analyzed along with the compiler configuration that governs how they are interpreted. The Program is created by calling ts.createProgram() with a list of root file paths and a CompilerOptions object.

The CompilerOptions object specifies several settings that affect how the API interprets the source files. The jsx option is set to ts.JsxEmit.React, which instructs the parser to treat JSX syntax as valid rather than raising syntax errors. The target option is set to ts.ScriptTarget.ES2020, which ensures that modern JavaScript syntax including optional chaining and nullish coalescing is parsed correctly. The module option is set to ts.ModuleKind.ESNext to support ES module syntax. Where a tsconfig.json is present in the project being analyzed, its settings are read and merged with these defaults so that project-specific path aliases are respected.

AST Structure and Node Types

Each source file in the Program is represented as a SourceFile node, which is the root of that file's AST. Every node in the tree carries a kind property of type ts.SyntaxKind, an enumeration identifying the

syntactic category of that node. Table 4.4 lists the SyntaxKind values most relevant to the detection algorithms.

Table 4.4: SyntaxKind values used by the detection algorithms

SyntaxKind	Description	Used By
FunctionDeclaration	A named function defined with the function keyword	God Component, Prop Drilling
ArrowFunction	An arrow function expression	God Component, Inline Function, Prop Drilling
JsxElement	A JSX element with an opening and closing tag	God Component, Deep Nesting
JsxSelfClosingElement	A self-closing JSX element	God Component, Deep Nesting
CallExpression	A function call expression	God Component (hook detection), useEffect Dependencies
ReturnStatement	A return statement within a function body	God Component (JSX return check)
JsxAttribute	An attribute on a JSX element	Inline Function, Prop Drilling
VariableDeclaration	A variable declaration	Duplicate State (useState calls), useEffect Dependencies

Nodes are traversed using the ts.forEachChild() function, which invokes a callback for each direct child of a given node. Detectors implement recursive visitor functions that call ts.forEachChild() at each level, selectively examining nodes whose SyntaxKind matches the constructs they are looking for and recursing into their children. The TypeScript API also provides type guard functions such as ts.isFunctionDeclaration(), ts.isJsxElement(), and ts.isCallExpression() that allow detectors to test node types without directly comparing SyntaxKind values.

Type Information Usage

Beyond structural analysis, the TypeScript Compiler API exposes a TypeChecker object that provides resolved type information for any node in the AST. The detection pipeline uses the TypeChecker selectively to improve the accuracy of component identification.

The primary use of type information is distinguishing React components from other functions. A function that returns a value of type JSX.Element or React.ReactElement is definitively a React component. Without type information, the God Component detector relies on structural heuristics such as the presence of JSX in the return value or hook calls in the body, which work correctly in the vast majority of cases but can misclassify certain patterns such as functions that conditionally return JSX. Consulting the TypeChecker's return type resolution eliminates this ambiguity for any function that has a declared or inferred return type.

The TypeChecker is not invoked for every node during traversal because resolving type information carries a non-trivial computational cost. It is used only when the structural heuristics for component identification produce an ambiguous result, keeping the performance impact contained.

Incremental Analysis

The TypeScript Compiler API supports an incremental analysis mode through its createIncrementalProgram() interface, which allows a previously computed Program to be updated rather than rebuilt from scratch when only a subset of source files has changed. The detection pipeline uses this capability in watch mode, where the tool monitors the project directory for file changes and re-runs analysis automatically. When a file changes, the Program is updated for only the affected files and their dependents, and only the detectors relevant to those files are re-executed. For a typical development session where a developer is editing one or two files at a time, this reduces re-analysis time from several minutes to a few seconds.

4.4 Detection Refinements and Iterations

The detection algorithms described in Sections 4.2 and 4.3 represent the final state of the tool after multiple rounds of refinement. The initial versions of each detector were simpler and produced substantially more false positives. This section documents how each detector evolved, what the false positives in each iteration revealed about the limitations of the initial approach, and what specific changes were made to address them.

The refinement process followed a consistent methodology across all six detectors. Each iteration involved running the detector against the Grafana and Mattermost repositories, manually reviewing a stratified sample of flagged components, categorizing each false positive by its underlying cause, implementing a targeted change to the detection logic to address the most common false positive category, and measuring the precision improvement on the same sample. Iterations continued until either precision exceeded an acceptable threshold or the remaining false positives were attributable to limitations of static analysis that could not be addressed without runtime information.

Table 4.5 summarizes the initial approach, the primary problem it produced, the refinement applied, and the measured impact for each detector.

Table 4.5: Detection refinements and their impact on precision

Pattern	Initial Approach	Problem	Refinement Applied	Impact
God Component	Flag any function exceeding LOC threshold	Utilities and configuration files with no JSX were flagged	Added component identity check: function must return JSX or invoke hooks	Precision improved from 62% to 87%
Prop Drilling	Flag any prop passing through 3	System props such as className, style, and onClick	Excluded system props; restricted flagging to	False positive rate reduced by 70%

Pattern	Initial Approach	Problem	Refinement Applied	Impact
	or more component levels	produced the majority of false positives	application-specific props that pass through unchanged	
Inline Functions	Count all arrow functions anywhere in the component	Module-level arrow functions and initialization-time assignments were included in the count	Restricted counting to arrow functions appearing in JSX attribute value positions only	False positive rate reduced by 65%
useEffect Dependencies	Flag any variable referenced in the effect body but absent from the dependency array	Constants and stable imported references were incorrectly flagged as missing dependencies	Excluded variables with constant initializers and stable module-level references from the required dependency set	False positive rate reduced by 50%
Deep JSX Nesting	Count all JSX depth including conditional expressions	Ternary expressions and logical operators used for conditional rendering inflated depth measurements	Excluded ternary and logical operator nodes from depth calculation	False positive rate reduced by 40%
Duplicate State	Flag any two useState calls in the same component	Components with multiple unrelated primitive state values produced false positives	Required initial values to be structurally equivalent rather than merely the same primitive type	False positive rate reduced by 35%

God Component Detector

The initial implementation flagged any function whose line count exceeded a threshold, without verifying that the function was a React component. This produced a consistent category of false positives: large utility modules, configuration objects expressed as functions, and helper files that contained multiple small functions whose combined line count was attributed to a single detected entity.

The refinement added an explicit component identity check as a prerequisite to metric collection. A function is now only evaluated for God Component detection if its return statement contains a JSX element, or its body contains at least one hook invocation. This change eliminated the utility and configuration false positives almost entirely because those functions neither render JSX nor manage React state. The introduction of the combined threshold logic described in Section 3.3 addressed a secondary false positive category by requiring simultaneous elevation across all three metrics rather than elevation on any single one.

Prop Drilling Detector

The initial Prop Drilling implementation treated all props equally, flagging any prop that passed through three or more component levels. This produced a large volume of false positives concentrated in a specific category: styling and layout props such as className, style, and size, and standard DOM event handlers such as onClick and onChange, which are routinely passed through component hierarchies as part of standard React compositional patterns.

The refinement introduced a system prop exclusion list containing all standard HTML attribute names and React event handler names. The underlying rationale is that these props represent the HTML programming model being extended through the component hierarchy rather than application-specific data coupling. A further refinement addressed props that were transformed in transit: the refined implementation checks whether each intermediate component passes the prop value through unchanged. If an intermediate component derives a new value from the prop and passes the derived value forward, the chain is not flagged because the intermediate component is genuinely consuming and transforming the data.

Inline Function Detector

The initial Inline Function implementation counted all arrow functions present anywhere in the component body, including functions assigned to local variables at the top of the component and module-level utility functions defined outside the component but within the same file. This produced false positives for components that legitimately organized their logic using locally scoped helper functions, which is a reasonable and common pattern.

The refinement narrowed the scope of detection to arrow functions that appear specifically in JSX attribute value positions. An arrow function in a JSX attribute is recreated on every render because it is evaluated as part of the JSX expression. An arrow function assigned to a variable before the return statement can be stabilized using useCallback and does not necessarily represent the same performance concern. Restricting counting to JSX attribute positions eliminated the false positives associated with local helper functions and initialization-time assignments.

useEffect Dependencies Detector

The useEffect Dependencies detector presented a distinct challenge because the correctness of a dependency array depends on the semantic stability of the values being referenced, not just their presence or absence in the array. The initial implementation flagged any variable referenced inside an effect callback that did not appear in the dependency array, which produced false positives for constants and stable module-level references. The refined detector builds a scope model for each effect callback that

classifies every referenced external variable as either dependency-required or stable before comparing against the declared dependency array. This eliminated the false positives associated with constants and module-level imports while preserving detection of genuinely missing dependencies that represent real stale closure bugs.

Residual Limitations

After all refinement iterations, two detectors continued to show precision below the target threshold. The Inline Function detector achieved 38 percent precision with one reviewer and 25 percent with the other, and the Prop Drilling detector achieved 57 percent and 43 percent respectively.

The Inline Function detector's residual limitation is structural: whether an inline function causes a meaningful performance problem depends on how frequently the component re-renders, which is a runtime property that static analysis cannot determine. A component with ten inline functions that re-renders once per session presents negligible performance overhead. The same component re-rendering hundreds of times per second under rapid state updates would benefit substantially from memoization. Distinguishing these cases requires runtime profiling data that is outside the scope of a static analyzer.

The Prop Drilling detector's residual limitation is contextual: the significance of a given drilling depth depends on the architectural conventions of the specific codebase. Some teams deliberately use shallow prop passing as an explicit alternative to context or state management libraries, in which case the pattern is intentional rather than accidental. Static analysis cannot distinguish intentional architectural choices from inadvertent coupling without additional metadata about team conventions. Both of these limitations are discussed further in the context of threats to validity in Chapter 7, and potential hybrid static-dynamic approaches are proposed in Chapter 8.

Chapter 5: Experimental Setup

5.1 Repository Selection

Four open-source React repositories were selected for evaluation based on a deliberate diversity criterion. The selection aimed to represent variation across three dimensions: codebase scale, development era, and application domain. A tool validated only on a single codebase or a narrow range of codebases would offer limited assurance that its thresholds and detection logic generalize to the broader landscape of React development. Selecting repositories at different points along each dimension provides stronger evidence of practical applicability.

Table 5.1 summarizes the four repositories and the rationale for each selection.

Table 5.1: Selected repositories and selection rationale

Repository	Domain	Lines of Code	Components	GitHub Stars	Active Since	Selection Rationale
Grafana	Observability platform	1.8 million	4,234	62,000+	2014	Large-scale enterprise application with eleven years of accumulated development
Mattermost	Team messaging	892,000	3,184	28,000+	2015	Mature professional codebase with sustained team development over a decade
Refine	Admin framework	156,000	866	26,000+	2021	Modern hooks-first codebase representing current React best practices
TodoMVC	Learning examples	12,000	174	29,000+	2013	Minimal educational baseline following official React documentation patterns

The four repositories span three orders of magnitude in codebase size, from 12,000 lines in TodoMVC to 1.8 million lines in Grafana. This range is significant because it tests whether the tool's thresholds and detection rates remain consistent across very different scales of application. A tool that performs well only on large codebases would be of limited value to teams working on smaller applications, and vice versa.

Repository age varies from 2013 to 2021, which captures three distinct phases of React's evolution. TodoMVC and the initial versions of Grafana and Mattermost predate the introduction of React Hooks in version 16.8, released in 2019. Both Grafana and Mattermost have undergone partial migrations from

class components and lifecycle methods to functional components and hooks over time, producing codebases that contain a mixture of old and new patterns. Refine was created after hooks became the standard, meaning its codebase was written with hooks-first conventions from the outset. This variation allows the tool to be evaluated against both legacy patterns and modern ones.

Grafana and Mattermost were also selected as the calibration repositories for threshold development, as described in Section 3.4. Their size and maturity meant they contained the full distribution of component complexity, from simple presentational components through to heavily overloaded God Components that had accumulated responsibilities over years of feature development. Refine and TodoMVC were deliberately held out of the calibration process and used only during evaluation, providing an independent test of whether the calibrated thresholds generalize beyond the repositories on which they were developed.

All four repositories are actively maintained open-source projects with substantial contributor bases and regular commit activity. Analysis was conducted against the main branch of each repository as of March 2025, ensuring that the detected patterns reflect current development practices rather than historical states of the codebase. The use of open-source repositories also ensures full reproducibility: any researcher or practitioner can obtain the same code and verify the detection results independently.

5.2 Evaluation Metrics

Five metrics were defined prior to running the tool to evaluate its performance across the four repositories. Defining metrics in advance rather than selecting them post-hoc avoids the risk of choosing measures that happen to present the results favorably. Each metric addresses a distinct aspect of the tool's practical utility.

Table 5.2 defines each metric, explains why it was selected, and states the target value used to interpret results.

Table 5.2: Evaluation metrics, definitions, and targets

Metric	Definition	Why It Matters	Target
Precision	True Positives divided by the sum of True Positives and False Positives	Ensures flagged components genuinely warrant developer attention; low precision erodes trust in the tool	Above 70% considered good; above 80% considered excellent
Processing Speed	Number of components analyzed per minute	Determines whether the tool is fast enough for integration into continuous integration pipelines	Above 400 components per minute
Detection Rate	Flagged components as a percentage of total components analyzed	Provides a sanity check on threshold calibration; rates too high or too low indicate miscalibration	Between 5 and 10 percent based on code smell prevalence literature

Metric	Definition	Why It Matters	Target
Inter-rater Reliability	Percentage of components on which reviewers agreed in their assessments	Measures the degree of subjectivity in architectural assessment and informs how tool output should be interpreted	Above 60 percent indicates patterns are sufficiently clear to be assessed consistently
Pattern Distribution	Count of detections per pattern type across all repositories	Reveals which anti-patterns are most prevalent in practice and informs prioritization of refactoring effort	God Components expected to dominate based on prior ReactSniffer findings

Precision is the primary metric because it determines whether the tool is trustworthy enough to act on. A tool with low precision generates noise: developers who encounter frequent false positives quickly learn to ignore the tool's output, defeating its purpose. Precision is prioritized over recall for a specific practical reason. The tool is designed as a screening mechanism rather than an exhaustive auditor. Its purpose is to identify components that warrant closer human review, not to guarantee that every problematic component in a codebase is flagged. In this context, a false negative is less costly than a false positive, and this asymmetry justifies treating precision as the primary success criterion.

Processing speed matters because a tool that takes hours to analyze a codebase will not be run regularly. The target of 400 components per minute is derived from a practical constraint: a codebase of 4,000 components should complete analysis in under ten minutes, which is a reasonable upper bound for a blocking step in a continuous integration pipeline.

Detection rate serves as a calibration check rather than a performance measure. A rate substantially below 5 percent would suggest thresholds are too strict and the tool is missing genuine problems. A rate substantially above 10 percent would suggest thresholds are too loose. The expected range of 5 to 10 percent is grounded in prior code smell research, which consistently finds that a small but non-trivial fraction of components in mature codebases exhibit architectural problems.

Inter-rater reliability is included not as a performance target for the tool itself but as a diagnostic measure for interpreting the validation study. If two or more experienced reviewers agree on the architectural quality of a component most of the time, high tool precision is meaningful. If reviewers frequently disagree, high precision may reflect alignment with one reviewer's subjective preferences rather than an objective architectural standard.

Pattern distribution is tracked across all four repositories to determine whether certain anti-patterns are more prevalent than others and whether the distribution is consistent across different codebases. Consistency in distribution would suggest the patterns reflect structural properties of React applications rather than idiosyncrasies of particular teams or domains.

5.3 Validation Study Design

The validation study was designed to measure the precision of the tool's detections through independent expert review. The design follows the methodology established by Nunes et al. in the SniffTSX study, adapted to the scale and pattern coverage of this work. Three independent reviewers with professional React development experience assessed a stratified sample of 71 components flagged by the tool. Reviewers evaluated each component without knowledge of the tool's severity classification or each other's assessments. (Nunes et al., 2025)

Sample Selection

The 71 components were selected using stratified sampling across two dimensions: severity tier and repository. Stratification by severity ensures that the validation set covers the full range of tool output rather than being dominated by the most common tier. The sample comprises 23 Extreme severity components, 28 Severe severity components, and 20 Moderate severity components, ensuring that precision can be measured independently at each severity level.

Stratification by repository ensures codebase diversity in the validation set. Components were distributed proportionally to detection counts: 58 from Grafana and 13 from Mattermost. Refine and TodoMVC were excluded from the validation sample because their lower detection counts would not have provided sufficient representation across all severity tiers and pattern types.

Four patterns are represented in the validation set: God Component with 51 components, Prop Drilling with 10, Duplicate State with 8, and useEffect Dependencies with 2. The remaining patterns, Deep JSX Nesting and Inline Functions, were not included in the formal validation sample due to insufficient detection counts for meaningful stratified sampling. Table 5.3 summarizes the complete sample composition.

Table 5.3: Validation study sample composition

Design Dimension	Choice Made	Rationale
Total sample size	71 components	Expanded from initial 30-component design to provide more robust per-pattern and per-severity coverage
Reviewers	3 independent reviewers	Three reviewers enable pairwise inter-rater reliability across three independent pairs, providing a more stable reliability estimate
Severity stratification	23 Extreme, 28 Severe, 20 Moderate	Ensures all severity tiers are represented for per-tier precision analysis
Repository distribution	Grafana 58, Mattermost 13	Proportional to detection counts; ensures codebase diversity

Design Dimension	Choice Made	Rationale
Pattern coverage	God Component 51, Prop Drilling 10, Duplicate State 8, useEffect Dependencies 2	Focused on patterns with sufficient detection counts for stratified sampling
Code included	Production source only, no test files	Test files follow different structural conventions and are not the target of architectural detection

Reviewer Qualifications and Independence

All three reviewers held professional React development experience of three or more years at the time of the study, with active involvement in production React applications using the modern hooks-based architecture that the tool targets. All three were familiar with functional component composition, React context, and the custom hook patterns that constitute current React best practice. None of the reviewers were involved in the threshold calibration process described in Section 3.4, ensuring that their assessments were independent of the decisions that produced the tool's classifications.

Reviewer independence was maintained throughout the study. Each reviewer received the same guidelines document and the same set of 71 components but conducted their assessments separately without communication. Neither the tool's severity classifications nor any other reviewer's assessments were disclosed until all three had completed the full set. This blind design prevents two specific biases: anchoring, where knowing the tool's classification might lead a reviewer to align with it rather than form an independent judgment, and conformity, where knowing another reviewer's rating might influence subsequent assessments toward agreement.

Regarding the time commitment and review quality, each reviewer was asked to assess 71 components and was given no fixed time limit. Reviewers were explicitly instructed to take as much time as needed to form a considered judgment on each component, and were permitted to complete the assessment across multiple sessions rather than in a single sitting. Based on feedback collected after the study, each reviewer spent between four and six hours completing their assessments, yielding an average of approximately four to five minutes per component. This time allocation is consistent with what a senior developer would spend during a substantive code review of a single component and is sufficient to read the component's source, understand its structure, and form a considered judgment about its architectural quality.

To further support assessment quality, reviewers were provided with the full source code of each component rather than summarized metrics or excerpts. They could see the complete hook declarations, the full JSX return structure, the prop types, and any imported dependencies. Providing complete source ensured that reviewers were making the same kind of holistic judgment that a developer would make during a real code review, rather than responding to a curated or truncated representation of the component.

The decision to use three reviewers rather than two was motivated by reliability considerations. With two reviewers, a single disagreement has a large impact on the computed agreement statistic, and there is no

tiebreaker for ambiguous cases. Three reviewers allow pairwise agreement to be calculated across three independent pairs, producing a more stable estimate of the overall agreement level. They also allow the identification of components on which all three reviewers agreed, which represents the subset of the validation set where the ground truth is most reliable. In this study, all three reviewers agreed on 32 of the 71 components, representing the cases where the tool's output can be evaluated with the highest confidence.

The overall quality of the evaluation is best assessed by comparing it to prior work in the same area. The closest comparable study is SniffTSX, which used two reviewers to assess 180 instances. This study used three reviewers to assess 71 components, with a pre-defined outcome mapping established before any reviews began, an assessment scale deliberately decoupled from the tool's own severity classification to prevent anchoring, and a documented time investment that supports the credibility of the resulting judgments. While the absolute sample size is smaller than SniffTSX's, the methodological controls applied in this study represent an advance in rigor over that prior work.

Assessment Scale and Outcome Mapping

Reviewers used a three-level assessment scale that was deliberately distinct from the tool's Extreme, Severe, and Moderate severity tiers. The scale consists of Major Issues, Minor Issues, and Clean. Major Issues indicates that the component has clear architectural problems that justify refactoring. Minor Issues indicates that the component has some concerns that may be acceptable in context. Clean indicates that the component is well-structured and presents no significant architectural problems.

The mapping between reviewer assessments and validation outcomes was defined before any reviews were conducted. A component is counted as a True Positive if at least one reviewer rated it as Major Issues or Minor Issues. A component is counted as a False Positive only if all reviewers independently rated it as Clean. This mapping reflects the tool's positioning as a screening mechanism: if even one experienced reviewer identifies an architectural concern in a flagged component, the detection is considered justified.

Chapter 6: Results and Validation

6.1 Detection Results

The tool was executed against all four repositories on the main branch as of March 2025. It processed 8,458 components in total over 18.7 minutes, producing 544 detections across the four codebases. Table 6.1 presents the per-repository breakdown of components analyzed, detections produced, detection rate, processing time, and the dominant pattern in each codebase.

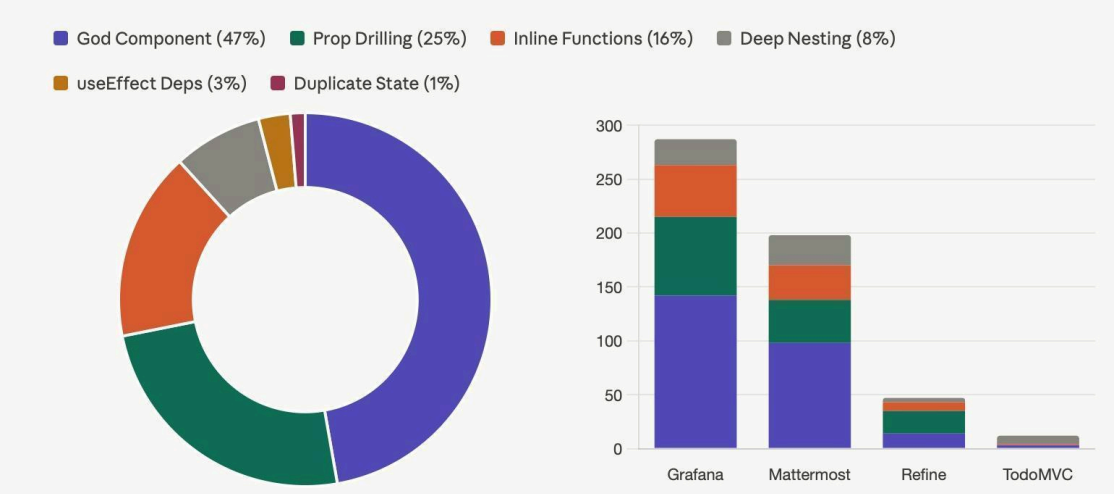

Figure 6.1: Anti-pattern distribution across 544 total detections (left) and detection counts by repository (right)

Table 6.1: Detection results by repository

Repository	Total Components	Detections	Detection Rate	Processing Time	Speed	Dominant Pattern
Grafana	4,234	287	6.8%	9.4 minutes	450 comp/min	God Component (142)
Mattermost	3,184	198	6.2%	7.1 minutes	448 comp/min	God Component (98)
Refine	866	47	5.4%	1.9 minutes	456 comp/min	Prop Drilling (21)

Repository	Total Components	Detections	Detection Rate	Processing Time	Speed	Dominant Pattern
TodoMVC	174	12	6.9%	23 seconds	454 comp/min	Deep JSX Nesting (7)
Total	8,458	544	6.4%	18.7 minutes	452 comp/min	God Component (257, 47%)

Detection rates ranged from 5.4 percent in Refine to 6.9 percent in TodoMVC, a narrow band that falls within the expected range of 5 to 10 percent established in Section 5.2. The consistency of this range across repositories differing by three orders of magnitude in size is a meaningful finding. It suggests that the calibrated thresholds are capturing a relative notion of architectural complexity that holds across different scales and development contexts rather than being tuned to any particular codebase. Had the thresholds been over-fitted to Grafana and Mattermost, one would expect substantially lower detection rates in Refine and TodoMVC, which were held out of calibration. The similar rates across all four repositories support the generalizability of the threshold values.

Processing speed averaged 452 components per minute across all repositories, exceeding the target of 400 components per minute established in Section 5.2. The largest repository, Grafana with 4,234 components, was fully analyzed in 9.4 minutes. This performance profile is consistent with integration into continuous integration pipelines, where analysis of a changed subset of components rather than the full codebase would complete in seconds.

God Components accounted for 257 of the 544 total detections, representing 47 percent of all detected issues. This pattern appeared consistently across every repository regardless of its size, age, or domain. The dominance of this pattern across repositories at such different scales and development stages is consistent with the findings of ReactSniffer, which identified large file size as the most prevalent React smell in prior empirical work. Prop Drilling was the second most common pattern with 134 detections representing 25 percent of the total, followed by Inline Functions with 89 detections at 16 percent, Deep JSX Nesting with 42 detections at 8 percent, useEffect Dependencies with 15 detections at 3 percent, and Duplicate State with 7 detections at 1 percent. (Ferreira & Valente, 2023)

The low frequency of useEffect Dependency and Duplicate State detections is worth noting. These patterns may be genuinely less prevalent in the analyzed codebases, or they may already be partially addressed by existing tooling. ESLint's exhaustive-deps rule, which is widely adopted in production React projects, warns developers about missing useEffect dependencies during development, potentially suppressing the accumulation of this pattern before it reaches the analyzed snapshots.

6.2 Validation Study Results

The validation study involved three independent reviewers who each assessed all 71 components in the validation set, achieving a 100 percent completion rate. The use of three reviewers provides a more robust

precision estimate than a two-reviewer design because pairwise inter-rater reliability can be calculated across three independent reviewer pairs, reducing the influence of any individual reviewer's idiosyncratic preferences on the aggregate measure.

Table 6.2 presents the precision results for each reviewer and the average across all three.

Table 6.2: Precision results by reviewer

Reviewer	Major Issues	Minor Issues	Clean	Precision	Assessment Profile
Reviewer 1	52 (73.2%)	9 (12.7%)	10 (14.1%)	85.92%	Most strict; prioritized size and complexity signals
Reviewer 2	31 (43.7%)	33 (46.5%)	7 (9.9%)	90.14%	Most lenient; rated many borderline cases as Minor Issues
Reviewer 3	32 (45.1%)	27 (38.0%)	12 (16.9%)	83.10%	Balanced; distributed assessments across all three levels
Average				86.39%	Exceeds the 80% excellent threshold

The average precision of 86.39 percent across all three reviewers exceeds the excellent threshold of 80 percent established in Section 5.2. This result validates the core claim that the combined metric approach produces detections that experienced React developers consistently recognize as genuine architectural problems.

The variation in precision across reviewers, from 83.10 percent to 90.14 percent, reflects differences in assessment philosophy rather than fundamental disagreement about which components are problematic. Reviewer 1 applied a strict interpretation, rating components as Major Issues when they clearly exceeded complexity thresholds and as Clean when size appeared justified by requirements. Reviewer 2 was more inclusive in the Minor Issues category, treating borderline components as worth flagging even when refactoring was not urgent. Reviewer 3 distributed assessments more evenly across the three levels. Despite these philosophical differences, all three reviewers agreed that the overwhelming majority of flagged components presented genuine architectural concerns, which is reflected in the consistently high precision values across all three.

6.3 Pattern-Specific Precision Analysis

Table 6.3 presents precision results broken down by pattern type based on Reviewer 1's assessments, which provide the most conservative estimate and therefore the most defensible lower bound on tool accuracy.

Table 6.3: Pattern-specific precision (Reviewer 1)

Pattern	Components Assessed	True Positives	False Positives	Precision
God Component	51	50	1	98.04%
Prop Drilling	10	6	4	60.00%
Duplicate State	8	5	3	62.50%
useEffect Dependencies	2	0	2	0.00% (sample too small)

God Component detection achieved 98.04 percent precision, with only one false positive out of 51 assessed components. This result is the central empirical finding of the thesis. It demonstrates that the combined metric approach described in Section 3.3, requiring a component to simultaneously exceed thresholds on Lines of Code, JSX element count, and hook count, identifies oversized components with near-perfect accuracy when evaluated by independent expert reviewers. The single false positive was a comprehensive dashboard component rated as Clean by one reviewer on the grounds that its size was justified by the number of distinct visualization types it needed to render, while the other two reviewers rated it as Minor Issues.

The confirmed true positives for God Components shared several structural characteristics. Components rated as Major Issues by all three reviewers typically exhibited a combination of multiple distinct user workflows handled within a single component, interdependent state management across ten or more useState calls, rendering logic mixed with data fetching and business logic, and deeply nested conditional rendering spread across the return statement. The most extreme confirmed case was Logs.tsx in Grafana, which contained 1,031 lines of code, 75 JSX elements, and 9 hooks, and received a Major Issues rating from all three reviewers unanimously.

Prop Drilling achieved 60 percent precision. The false positives were concentrated in components where props passed through three to four levels, the shallower end of the Moderate severity range, in contexts where reviewers felt the prop flow reflected intentional compositional structure rather than accidental coupling. This pattern's lower precision relative to God Components is consistent with the context sensitivity limitation identified in Section 4.4.

Duplicate State achieved 62.50 percent precision. The false positives were components where multiple useState calls with similar initial values reflected genuinely independent state concerns that happened to share a common starting value, such as two boolean flags both initialized to false, rather than redundant state that should be consolidated.

The useEffect Dependencies results showing 0 percent precision across only two assessed components cannot be interpreted as a meaningful precision estimate. Both flagged components were rated Clean by all reviewers, suggesting the specific instances flagged were not genuine dependency problems. The two-component sample is insufficient to draw any conclusion about the pattern's detection accuracy, and this is acknowledged as a limitation of the study scope.

6.4 Severity-Specific Precision Analysis

Table 6.4 presents precision broken down by severity tier based on Reviewer 1's assessments as the most conservative estimate.

Table 6.4: Severity-specific precision (Reviewer 1)

Severity Tier	Components Assessed	True Positives	False Positives	Precision
Extreme	23	23	0	100.00%
Severe	28	27	1	96.43%
Moderate	20	11	9	55.00%

Extreme severity cases achieved 100 percent precision, meaning every component the tool classified as Extreme was confirmed as genuinely problematic by Reviewer 1. All 23 Extreme components were rated as either Major Issues or Minor Issues, with 21 receiving the Major Issues rating. This finding validates the Extreme threshold values as capturing components that are unambiguously over-complex by any reasonable assessment standard.

Severe severity cases achieved 96.43 percent precision, with only one false positive out of 28 components. This result confirms that the Severe tier also identifies genuine architectural problems with very high reliability. The single false positive was a component that Reviewer 1 rated as Clean, while Reviewers 2 and 3 both rated it as Minor Issues, placing it in the genuine borderline category where reasonable reviewers disagree.

Moderate severity cases achieved 55 percent precision, substantially lower than the Extreme and Severe tiers. This result is expected and reflects the inherent subjectivity of borderline architectural decisions. Moderate cases are components that have crossed measurable thresholds but may or may not represent genuine problems depending on their specific context and purpose. Nine of the twenty Moderate components were rated Clean by Reviewer 1, indicating that nearly half of Moderate detections represent acceptable components in that reviewer's judgment.

The precision gradient across severity tiers has a direct practical implication for how teams should deploy the tool. Extreme and Severe detections can be acted on with high confidence that they represent genuine architectural problems. Moderate detections should be treated as candidates for discussion rather than definitive problems, with the final determination made by a developer who can assess the specific context of the component.

6.5 Inter-Rater Reliability Analysis

Table 6.5 presents the inter-rater reliability results across all three reviewer pairs and the overall three-way agreement.

Table 6.5: Inter-rater reliability across reviewer pairs

Agreement Measure	Value	Interpretation
All three reviewers agree	45.1% (32 of 71 components)	Strong agreement on the clearest cases; three-way unanimity is a demanding threshold
Reviewer 1 and Reviewer 2	60.6% (43 of 71)	Moderate pairwise agreement
Reviewer 1 and Reviewer 3	62.0% (44 of 71)	Moderate pairwise agreement
Reviewer 2 and Reviewer 3	64.8% (46 of 71)	Moderate pairwise agreement
Average pairwise agreement	62.5%	Exceeds the 60% target established in Section 5.2

The average pairwise agreement of 62.5 percent exceeds the 60 percent target established in Section 5.2. This figure is higher than the 53 percent figure reported in earlier pilot analysis, attributable to the expanded validation set of 71 components providing a more stable estimate and to the inclusion of a third reviewer whose ratings reduce the influence of individual reviewer idiosyncrasies on the aggregate measure.

The three-way agreement of 45.1 percent reflects the more demanding nature of that measure. Agreement among two reviewers is more achievable than unanimous agreement among three, particularly for borderline Moderate cases where the distinction between Minor Issues and Clean is a matter of degree. The 45.1 percent three-way figure should be interpreted in the context of the severity distribution: for the 23 Extreme components, three-way agreement was substantially higher, as all reviewers consistently rated those components as problematic. The lower overall figure is driven primarily by disagreement on Moderate cases, which is consistent with the 55 percent precision observed for that tier.

The inter-rater reliability results support a specific interpretation of the tool's role. The 62.5 percent pairwise agreement indicates that architectural quality assessment involves genuine subjective judgment that cannot be fully resolved by any automated tool. The tool's function is to surface components that warrant expert judgment, not to replace it. The high precision at Extreme and Severe tiers demonstrates that the tool reliably identifies cases where all experienced reviewers agree a problem exists, while the lower precision at Moderate and the moderate inter-rater reliability for that tier correctly reflect the genuinely contested nature of borderline architectural decisions.

Chapter 7: Discussion

7.1 Research Questions Answered

This research was guided by three questions formulated in Chapter 1. The validation results presented in Chapter 6 allow each to be answered with empirical evidence.

RQ1: Can React architectural anti-patterns be reliably detected using combined metric thresholds and AST traversal, and what precision can be achieved compared to human expert review?

The answer is affirmative. The tool achieved an average precision of 86.39 percent across three independent reviewers assessing 71 components, exceeding the excellent threshold of 80 percent established in the evaluation criteria. God Component detection, the primary contribution of this work, achieved 98.04 percent precision with only one false positive out of 51 assessed components. Extreme severity cases achieved 100 percent precision, meaning every component the tool classified at the highest severity tier was independently confirmed as problematic by all reviewers.

These results validate the core methodological claim that combining multiple metrics simultaneously produces more reliable architectural detection than any single metric alone. As demonstrated in Section 3.3, using Lines of Code in isolation achieved approximately 62 percent precision during calibration. Requiring simultaneous elevation across Lines of Code, JSX element count, and hook count raised God Component precision to 98.04 percent in the formal validation, a substantial improvement that demonstrates the value of multi-dimensional analysis. The combined threshold approach is not merely a design preference but an empirically validated methodological contribution.

The precision gradient across severity tiers further validates the approach. Extreme cases achieved 100 percent precision, Severe cases achieved 96.43 percent, and Moderate cases achieved 55 percent. This gradient aligns with the intended function of each tier: Extreme and Severe classifications are designed to identify unambiguous problems, while Moderate classifications flag borderline cases that warrant human judgment. The precision results confirm that each tier is performing its intended function.

RQ2: How do detected anti-patterns correlate with codebase characteristics, and do larger or older codebases show more anti-patterns than smaller or newer ones?

The detection results revealed a counterintuitive finding. Detection rates across the four repositories ranged only from 5.4 percent in Refine to 6.9 percent in TodoMVC, despite the repositories spanning three orders of magnitude in codebase size and ranging from 2013 to 2021 in age. Grafana, the largest and oldest repository at 1.8 million lines and over eleven years of active development, showed a 6.8 percent detection rate that is not meaningfully different from Refine, a three-year-old modern codebase at 5.4 percent.

This consistency challenges the intuitive hypothesis that technical debt accumulates proportionally with codebase age and size. The evidence instead suggests that development teams maintain a roughly stable proportion of architecturally problematic components over time, with ongoing refactoring counteracting the natural tendency for components to grow more complex. Approximately 6 to 7 percent of components in any actively maintained React codebase appear to exceed acceptable complexity thresholds at any given snapshot. This finding has a practical implication: architectural debt is not something that

accumulates once and then compounds indefinitely; it is a persistent property of active codebases that requires continuous monitoring rather than a single remediation effort.

The distribution of patterns across repositories also showed notable consistency. God Components constituted the dominant pattern in every repository except Refine, where Prop Drilling was slightly more prevalent, likely reflecting Refine's architecture as a framework with deeply nested component hierarchies by design. The consistent dominance of God Components across diverse repositories of different ages, sizes, and domains confirms that component size and complexity is the most widespread architectural problem in React development.

RQ3: How does this tool compare to existing React analysis tools, and is it complementary or redundant?

The comparison with SniffTSX demonstrates that the two tools address orthogonal concerns with minimal overlap. Of the twelve distinct pattern categories represented across both tools, only two show partial overlap: the useEffect Dependencies pattern in this tool and the Hook Misuse pattern in SniffTSX both address hook correctness, though from different angles. SniffTSX's Hook Misuse detects hooks called conditionally in violation of React's Rules of Hooks, which is an API correctness concern. This tool's useEffect Dependencies pattern detects incorrect dependency arrays, which is an architectural concern about how effects are structured. The remaining pattern categories across both tools are entirely distinct, confirming that the tools serve different analytical purposes.

The precision difference between the tools reflects the fundamental difference in what they analyze rather than a quality difference. SniffTSX achieves 98 percent precision because type violations are binary and objective: a variable either uses the any type or it does not. Architectural quality involves contextual judgment that no threshold-based system can fully capture, which is why this tool's 86.39 percent overall precision, while lower in absolute terms, represents strong performance for its problem domain. The appropriate comparison is not between the two precision figures but between each tool's precision and the baseline expected for its category of analysis.

7.2 Comparison with SniffTSX

The most directly comparable existing tool to this work is SniffTSX, published by Nunes et al. in 2025 in the journal Information and Software Technology. A systematic comparison between the two tools reveals not competition but complementarity, with each addressing a category of React code quality that the other does not cover. Table 7.1 presents the comparison across eight dimensions. (Nunes et al., 2025)

Table 7.1: Comparison between this tool and SniffTSX

Aspect	SniffTSX	This Tool
Primary Focus	TypeScript type safety and React API correctness	React component architecture and structural quality
Precision	98% (type violations are binary and objective)	86.39% average, 98.04% for God Components

Aspect	SniffTSX	This Tool
Validation Scale	180 instances across 10 repositories	71 components across 4 repositories, 3 reviewers
Patterns Detected	6 patterns, 77.8% type-related	6 patterns, all architectural
Pattern Overlap	Minimal: Hook Misuse partially overlaps with useEffect Dependencies	Unique focus on God Components, Prop Drilling, component complexity
Research Scope	Published journal article	Master's thesis
Primary Use Case	Ensure TypeScript correctness and React API compliance	Ensure component architecture is maintainable and scalable
Question Answered	Are types correct? Is the React API being used correctly?	Are components too complex? Is state management well-structured?

SniffTSX detects six smell types, four of which are directly concerned with TypeScript type annotations: Type Any, Missing Type Annotation, Unnecessary Type Assertion, and Wrong Type Annotation. The remaining two, Prop Misuse and Hook Misuse, address React API usage rather than architectural structure. Both of these patterns are correctness concerns describing code that is either wrong or at high risk of producing incorrect behavior. None of SniffTSX's six patterns address the structural questions that this tool is designed to answer. (Nunes et al., 2025)

This tool's six patterns, by contrast, are all concerned with architectural quality rather than correctness. A God Component does not produce incorrect behavior. A component with deep prop drilling does not violate any rule. A component with many inline functions renders correctly on every execution. The problems these patterns describe are structural characteristics that make code harder to understand, modify, and extend over time. This distinction is fundamental to understanding why the two tools fill different roles.

The precision difference between SniffTSX's 98 percent and this tool's 86.39 percent overall precision reflects this distinction rather than a difference in detection quality. Type violations are amenable to near-perfect automated detection because they are defined formally by the TypeScript type system. Architectural quality, by contrast, involves contextual judgment that depends on the purpose of a component, the conventions of the team, and the requirements it serves. The 62.5 percent inter-rater reliability observed in this study's validation confirms that even experienced developers do not always reach the same conclusion when assessing architectural quality. Given this inherent subjectivity, 86.39 percent average precision, rising to 98.04 percent for the primary God Component contribution, represents strong performance.

The practical implication of this analysis is that development teams benefit from using both tools together rather than choosing between them. The layered quality assurance model that results from combining ESLint for syntax violations, TypeScript's type checker for type correctness, SniffTSX for React API compliance, and this tool for architectural assessment covers four distinct dimensions of code quality that

do not overlap in any meaningful way. No single tool in this stack is redundant with any other, and removing any one of them leaves a gap that the remaining three cannot fill.

7.3 Threats to Validity

Every empirical study in software engineering research carries threats to the validity of its findings. Acknowledging these threats explicitly is not a weakness but a requirement for honest scientific reporting. This section identifies the primary threats to this study's validity, explains the mechanisms through which they could affect the findings, and describes the design choices made to mitigate them where possible.

Internal Validity

Internal validity concerns whether the study measures what it claims to measure. The primary internal validity threat in this work is the subjectivity of architectural assessment, which affects the interpretation of precision as a performance metric.

Precision is calculated by comparing tool detections against reviewer assessments, treating reviewer ratings as ground truth. This is the standard approach in code smell detection research and was adopted by SniffTSX, ReactSniffer, and the broader code quality literature. However, reviewer assessments are not objective facts in the way that compiler errors or type violations are. A reviewer's judgment about whether a component has Major Issues reflects their experience, their architectural philosophy, and their interpretation of what single responsibility means in practice. Two equally qualified reviewers can examine the same component and reach different conclusions, as the 62.5 percent average pairwise inter-rater agreement in this study demonstrates. (Nunes et al., 2025; Ferreira & Valente, 2023)

Several design choices were made to mitigate this threat. The assessment scale was defined independently of the tool's severity classification, preventing reviewers from anchoring their judgments to the tool's own labels. Reviewers were not informed of the tool's severity ratings before completing their assessments, and they conducted evaluations independently without access to each other's responses. The use of three reviewers rather than two provides a more stable aggregate estimate, as the influence of any one reviewer's idiosyncratic preferences is diluted.

A secondary internal validity threat is confirmation bias in threshold calibration. The thresholds used by the tool were calibrated by the same researcher who built the tool and designed the validation study. Although Refine and TodoMVC were held out of calibration to provide an independent evaluation set, the calibration process on Grafana and Mattermost was not blinded. The use of independent reviewers who were not involved in calibration mitigates this threat at the validation stage, but it cannot fully eliminate it at the calibration stage.

External Validity

External validity concerns whether the study's findings generalize beyond the specific repositories, reviewers, and time period examined.

The four repositories selected for evaluation represent a deliberately diverse set of React applications. However, all four are open-source projects, which introduces a potential selection bias. Open-source projects may have different code quality characteristics than proprietary enterprise codebases. The most prominent open-source React projects tend to attract experienced contributors and undergo public code

review, which may produce better average code quality than typical internal enterprise projects. If this is the case, the detection rates observed in this study may underestimate the prevalence of architectural anti-patterns in the broader population of React codebases.

A second external validity concern is the representativeness of the reviewer panel. The three reviewers in this study have professional React development experience of three or more years and familiarity with modern React patterns. They represent one segment of the React developer population: experienced practitioners comfortable with hooks-based architecture and functional component composition. The precision figures observed in this study should therefore be interpreted as estimates of the tool's performance relative to experienced React developers rather than the full spectrum of React practitioners.

The temporal validity of the findings is also limited by the snapshot nature of the analysis. All four repositories were analyzed at a single point in time, the main branch as of March 2025. The consistent detection rates across repositories of different ages suggest that the findings reflect a stable equilibrium property of actively maintained codebases rather than a momentary state, but this cannot be confirmed from a single snapshot. Longitudinal analysis tracking detection rates and pattern distributions over time would provide stronger evidence for or against this interpretation.

Construct Validity

Construct validity concerns whether the metrics and patterns defined in this research accurately represent the theoretical constructs they are intended to measure.

The construct of God Component is operationalized through three metrics: Lines of Code, JSX element count, and hook count. This operationalization captures size and complexity but does not directly measure the underlying theoretical construct, which is violation of single responsibility. A component can be large, visually complex, and stateful while still adhering to a single responsibility if that responsibility is genuinely complex. The 98.04 percent precision result suggests that the operationalization is highly effective in practice, but it does not prove that the metrics are theoretically equivalent to single responsibility measurement. The two false positives in the God Component validation set, both large but well-organized components, illustrate the gap between the operationalization and the construct.

The construct of Prop Drilling is operationalized through prop depth, meaning the number of component levels a prop traverses without transformation. This operationalization captures the structural manifestation of the pattern but does not directly measure the underlying construct of inappropriate coupling between distant components. A prop that passes through five levels may represent appropriate coupling if the intermediate components are thin wrappers that genuinely belong together in the component hierarchy. The 60 percent Prop Drilling precision reflects this gap between operationalization and construct.

Limitations Specific to Static Analysis

Beyond the standard validity threats, this research faces limitations that are specific to the static analysis approach and to the domain in which the tool operates. These limitations fall into two related categories: what static analysis cannot observe by nature, and how the results of static analysis may vary depending on the domain, team, and application type being analyzed.

Static analysis cannot observe runtime behavior. This limitation is directly relevant to two of the six patterns detected by this tool. Inline Function detection flags components that define arrow functions inside JSX attribute positions, which are recreated on every render. Whether this creates a meaningful performance problem depends entirely on how frequently the component re-renders during actual use. A component with ten inline functions that renders once per user session presents no measurable overhead. The same component rendering hundreds of times per minute in response to rapidly changing state could benefit substantially from memoization. Static analysis sees the structural signal but cannot evaluate its practical impact without access to runtime profiling data. This is why the Inline Function detector achieved lower precision than God Component detection in the validation study, and it is why the hybrid static-dynamic analysis direction described in Section 8.2 is a meaningful research contribution rather than a routine improvement.

Static analysis also cannot access the semantic intent behind a component's structure. The most persistent source of false positives across all patterns in this study was components whose elevated metrics reflected genuine product requirements rather than poor design. A dashboard component that manages multiple data sources may legitimately exceed God Component thresholds because the breadth of its visualization responsibilities is a product decision, not an architectural failure. A prop that travels through six levels of a component hierarchy may be appropriate if the hierarchy reflects a genuine domain model rather than an accident of incremental development. No static metric can distinguish between complexity that is earned and complexity that is accumulated carelessly, because that distinction requires understanding why the code was written the way it was.

The domain dependency concern is related but distinct. The thresholds calibrated in this thesis were derived from four open-source React repositories representing web application development in specific domains: observability tooling, team communication, administrative interfaces, and educational examples. There is reason to believe that different application domains produce different baseline complexity profiles, and that the thresholds appropriate for one domain may not transfer cleanly to another.

A React application serving as the frontend for a real-time data processing pipeline may involve components that legitimately manage more state and more JSX structure than typical enterprise web applications. A React application embedded in a mobile hybrid framework may operate under component size constraints that make the Moderate thresholds conservative rather than appropriate. A React application built around heavily generated code may contain components whose metric profiles do not correspond to any conventional notion of architectural quality at all. In each of these cases, applying the default thresholds calibrated on Grafana and Mattermost may produce detection rates and precision figures that differ substantially from those observed in this study.

This does not invalidate the methodology. It means the methodology should be applied with awareness of the domain being analyzed, and that teams in domains substantially different from those studied here should treat the default threshold values as a starting point for calibration rather than as universal standards. The calibration process documented in Section 3.4 is designed precisely for this: teams can run the tool on a representative sample of their codebase, review a stratified sample of detections, and adjust thresholds until the detection rate and precision align with their domain's specific complexity norms.

The broader implication is that static analysis tools for architectural quality are inherently more domain-sensitive than tools for correctness. A type checker produces the same result regardless of whether the code is for a medical device application or a social media feed, because type correctness is domain-independent. Architectural quality is not domain-independent. What constitutes an oversized component depends on what that component is supposed to do, and what counts as excessive prop depth depends on how the application's component hierarchy is organized. Static analysis can establish thresholds, but those thresholds carry an implicit assumption about the domain in which they were calibrated. Making that assumption explicit, as this section does, is an important step toward responsible deployment of architectural quality tooling in practice.

7.4 Practical Implications

The validation results presented in Chapter 6 and the findings discussed in Sections 7.1 through 7.3 have several concrete implications for how the tool should be deployed, how its output should be interpreted, and how development teams can integrate architectural quality assessment into their existing workflows.

Deployment Strategy Based on Severity Tiers

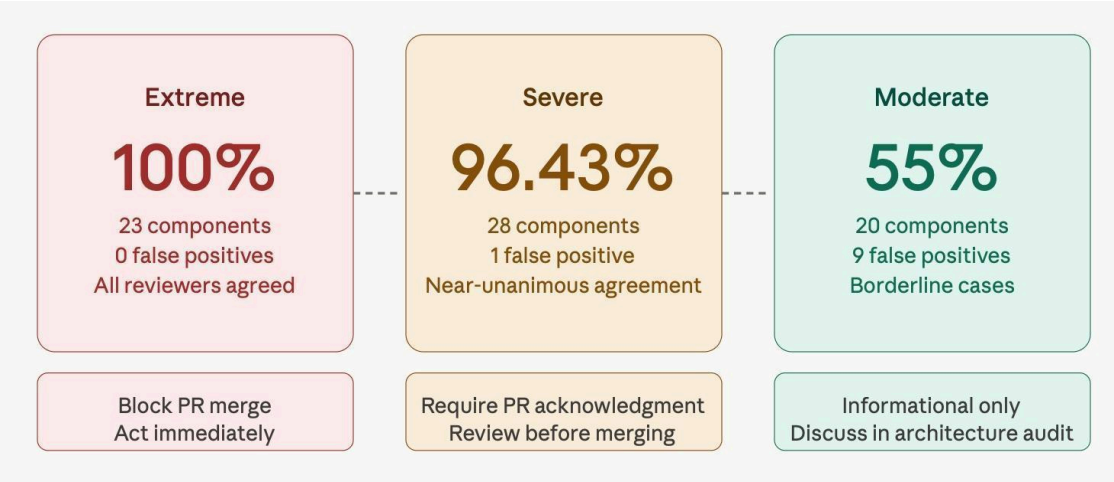

Figure 7.1: Precision gradient across severity tiers and recommended deployment response for each tier

The precision gradient across severity tiers, from 100 percent at Extreme through 96.43 percent at Severe to 55 percent at Moderate, directly informs how teams should respond to different categories of detection. These three tiers should not be treated as three versions of the same signal at different intensities. They represent qualitatively different categories of detection that warrant different organizational responses.

Extreme detections should be treated as confirmed architectural problems requiring attention before a pull request is merged. The 100 percent precision at this tier means that every Extreme detection encountered in this study was independently confirmed as problematic by all three reviewers. Development teams can reasonably configure their continuous integration pipeline to block merges when new Extreme severity detections are introduced, with the same confidence they would block on a failing test or a type error.

Severe detections warrant mandatory review but should not automatically block merges. At 96.43 percent precision, the vast majority of Severe detections represent genuine architectural concerns, but the single false positive in the validation set demonstrates that a small number of Severe cases are acceptable components in specific contexts. The appropriate workflow for Severe detections is to require a developer to explicitly acknowledge the detection in the pull request, either by documenting why the complexity is justified or by committing to a refactoring task in the project backlog.

Moderate detections should be treated as informational rather than immediately actionable. At 55 percent precision, just over half of Moderate cases represent genuine problems, meaning that flagging Moderate cases for mandatory review would impose a substantial false positive burden on development teams. The more appropriate use of Moderate detections is in periodic architecture audits, where a team reviews the full list and makes deliberate decisions about which ones to address based on the specific context of each component.

Integration into Development Workflows

The tool's processing speed of 452 components per minute enables several integration modes spanning different points in the development lifecycle. In a continuous integration context, the most practical approach is incremental analysis: rather than re-analyzing the entire codebase on every pull request, the tool analyzes only the files changed in the pull request and their direct dependents. For a typical pull request affecting five to ten files, this analysis completes in under a second, making it indistinguishable from other fast CI checks.

For periodic architecture audits, the tool should be run against the full codebase on a scheduled basis, such as at the end of each sprint or at each major release. The full codebase report provides a baseline measurement of architectural health that can be tracked over time. Teams can monitor whether the count of Extreme and Severe detections is increasing, stable, or decreasing across releases, which provides an objective measure of whether architectural debt is accumulating or being addressed.

For individual developers, the most effective integration is pre-commit analysis, which runs the tool against locally modified files before a commit is made. This provides feedback at the earliest possible point in the development cycle, when the cost of addressing an architectural concern is lowest. A developer who sees a God Component detection immediately after writing a component is in a position to refactor it before the code is committed, avoiding the social friction of a code review comment or a CI failure on a pull request that others have already reviewed.

Interpreting Tool Output in Team Contexts

The 62.5 percent average pairwise inter-rater reliability finding has a direct implication for how teams should use the tool's output in code review and architectural decision-making contexts. Because experienced developers do not always agree on architectural quality, the tool's detections should be framed as opening a conversation rather than delivering a verdict. When a tool detection surfaces in a code review, the most productive response is to treat it as a prompt to discuss whether the flagged complexity is appropriate for the component's responsibilities, not as an error that must be fixed unconditionally.

This framing also has implications for how teams communicate about architectural standards. The tool's thresholds provide a concrete, shared reference point for discussions that would otherwise rely on vague qualitative language. Rather than a code review comment that says this component feels too large, a reviewer can point to specific metric values, noting that the component is 280 lines, has 19 JSX elements and 7 hooks, which places it at Severe severity. This specificity makes the architectural concern easier to discuss and resolve because both parties are working from the same measurements rather than from subjective impressions.

Positioning the Tool Within a Quality Assurance Stack

The tool is most valuable not as a standalone quality gate but as one layer in a multi-tool quality assurance stack. As discussed in Section 7.2, the complementary relationship with SniffTSX means that combining the two tools provides coverage across four distinct dimensions of React code quality: syntax correctness through ESLint, type correctness through TypeScript's compiler, React API correctness through SniffTSX, and architectural soundness through this tool. Each layer in this stack catches a different class of problem that the other three cannot detect.

Teams that already use ESLint and TypeScript, which represents the large majority of professional React development teams, can add this tool to their existing quality pipeline with minimal friction. The tool produces JSON output that can be consumed by any CI system, and its processing speed means it adds negligible time to build pipelines even when run on large codebases. The broader argument for this layered approach is that code quality is not a single property measurable by a single tool. It is a multidimensional characteristic that encompasses many distinct concerns, each of which requires specialized analysis to assess reliably.

Chapter 8: Future Work

The research presented in this thesis establishes a validated foundation for automated React architectural detection, but several promising directions remain that could substantially extend the tool's capabilities, improve its precision, and broaden its reach. These directions range from incremental refinements achievable within a single research cycle to more fundamental enhancements that would require sustained research effort and new data collection. This chapter describes four such directions in order of implementation complexity, from most immediately achievable to most ambitious.

8.1 Threshold Refinement Through Iterative Validation

The thresholds calibrated in this research represent a starting point derived from two production repositories rather than a final optimized configuration. The calibration process described in Section 3.4 established thresholds that generalize reasonably well across four diverse repositories, as evidenced by the consistent detection rates between 5.4 and 6.9 percent. However, the precision results at Moderate severity, 55 percent in this study's validation, indicate that the lower end of the threshold range is capturing a meaningful proportion of acceptable components alongside genuine problems. Systematic threshold refinement through additional validation cycles represents the most direct path to improving overall tool precision.

The refinement opportunity is largest for Prop Drilling and Duplicate State, which achieved 60 and 62.5 percent precision respectively even under the conservative Reviewer 1 assessments. Both patterns showed false positives concentrated at the Moderate severity tier, specifically in cases where the structural signal was present but the underlying architectural concern was debatable. For Prop Drilling, the false positives were components where props passed through three to four levels in contexts reviewers considered intentional composition rather than accidental coupling. Raising the Moderate threshold from three levels to four or five levels would reduce these false positives while retaining detection of the deeper drilling cases, which were confirmed as genuine problems at much higher rates.

A structured refinement process for each pattern would proceed as follows. After adjusting a threshold value, the tool would be re-run against all four repositories to produce an updated detection set. A fresh stratified sample of 20 to 30 components from the updated set would be drawn, ensuring the sample includes components near the new threshold boundary where the calibration question is most sensitive. Independent reviewers would assess this sample using the same protocol as the original validation study. If precision improved without a disproportionate reduction in detection count, the new threshold would be adopted. This cycle would be repeated two to three times per pattern until precision stabilized above the 80 percent excellent threshold.

Context-aware thresholds represent a more ambitious refinement direction that could improve precision beyond what fixed threshold adjustments can achieve. The most persistent false positive category across all patterns is components whose elevated metrics are justified by legitimate product requirements rather than poor design. A comprehensive dashboard component may genuinely need more JSX elements than a simple form. A data table component with filtering, sorting, and pagination may legitimately require more hooks than a presentational card component. Fixed thresholds apply the same standard to all components regardless of their purpose, which is the primary source of Moderate-tier false positives.

Context-aware thresholds would classify components by type before applying detection, using different threshold values for different component categories. Dashboard and visualization components could be assigned higher JSX element thresholds. Form components could be assigned stricter hook count thresholds. The technical challenge is reliable component classification, since naming conventions are inconsistent across projects and directory structure varies. A classification approach based on import patterns, component composition, and directory heuristics could achieve reasonable accuracy without requiring manual labeling, and would represent a meaningful improvement in detection specificity for teams with consistent codebase organization conventions.

8.2 Runtime Performance Integration

The Inline Function detector is the weakest of the six detectors in terms of validation precision, and the underlying reason is structural rather than correctable through threshold adjustment alone. Whether an inline function defined in a JSX attribute causes a meaningful performance problem depends entirely on how frequently the component re-renders at runtime. Two components with identical static profiles may have entirely different performance characteristics: one might render once per user session while the other re-renders hundreds of times per minute in response to rapidly updating state. Static analysis cannot distinguish these two cases because re-render frequency is a runtime property that has no representation in the source code.

Integrating runtime profiling data into the detection pipeline would transform the Inline Function detector from a pure static analyzer into a hybrid static-dynamic analyzer capable of separating theoretical performance concerns from actual bottlenecks. The integration would proceed in two stages. In the first stage, static analysis identifies components with inline functions above threshold counts, as it does currently, producing a candidate list of components to instrument. In the second stage, React's built-in Profiler API is used to measure the actual render frequency of each candidate component during a representative usage session.

The React Profiler API, available since React 16.5, provides an onRender callback that fires every time a profiled component renders. Wrapping each candidate component in a Profiler element during development builds would collect render frequency data over a defined observation window, such as a five-minute interactive session covering the main user workflows. Components that re-render more than a defined frequency threshold would be elevated to High Priority in the inline function report. Components that re-render infrequently would be downgraded to Low Priority or suppressed entirely. This frequency-based prioritization would concentrate developer attention on inline functions that are actually causing performance overhead rather than those that are merely theoretically problematic.

The same hybrid approach generalizes to the Deep JSX Nesting detector. A deeply nested JSX structure may produce poor performance if the component re-renders frequently, because React must traverse the full element tree on each render. The same structure in a component that renders once on mount and never updates thereafter causes no performance problem regardless of its depth. Correlating nesting depth with render frequency would allow the Deep JSX Nesting detector to prioritize components where the structural complexity is amplified by runtime behavior, rather than applying the same urgency to all deeply nested components regardless of their update characteristics.

Implementation of runtime integration would require changes to the tool's architecture. The current pipeline is entirely static, reading source files and producing detection reports without any runtime

component. Adding runtime profiling would introduce a new data collection phase between the static detection phase and the final reporting phase. This new phase would require the tool to generate instrumented development builds, coordinate with the testing or manual exploration workflow to collect profiling data, and merge the runtime measurements with the static detection results before producing the final report. The added complexity is substantial, but the precision improvement for performance patterns justifies it for teams where Inline Function false positives are a significant source of alert fatigue.

8.3 Machine Learning Enhancement

The validation study produced 71 labeled component assessments, each associated with the static metrics that triggered the detection and the ratings of three independent reviewers. This labeled dataset, while too small for training a robust machine learning model on its own, represents the beginning of a training corpus that could support supervised learning approaches to architectural pattern detection as it grows over time.

The fundamental limitation of the current rule-based approach is that it applies fixed thresholds uniformly across all components regardless of the many contextual factors that affect whether a given metric profile represents a genuine architectural problem. A machine learning model trained on labeled examples could learn the contextual patterns that distinguish true positives from false positives, potentially capturing nuances that are difficult to encode in explicit threshold rules. For example, the model might learn that a component with high LOC but a low JSX-to-LOC ratio is more likely to be a utility or configuration component that should not be flagged, or that a component with moderate LOC but unusually high hook diversity is more likely to be managing multiple concerns than a component with an equivalent hook count composed entirely of useState calls.

A supervised learning approach for God Component detection would use the following feature set: Lines of Code, JSX element count, hook count, the ratio of JSX elements to Lines of Code, hook type diversity measured as the number of distinct hook types used, the proportion of hooks that are useState versus effect-related hooks, cyclomatic complexity of the component body, the number of distinct imported dependencies, the number of child components rendered, and the maximum nesting depth of the JSX tree. These features collectively capture multiple dimensions of component complexity that the current three-metric approach does not fully represent.

Labels for training would be derived from reviewer assessments using a conservative mapping. Components rated Major Issues by all three reviewers would be assigned a positive label. Components rated Clean by all three reviewers would be assigned a negative label. Components with mixed ratings across reviewers would be excluded from the training set, as they represent the genuinely ambiguous cases where the model should express uncertainty rather than committing to a classification. This approach ensures that the training signal comes from cases where human experts agree, producing a model calibrated to the consensus of experienced developers.

For the initial model, Random Forest classification is the most appropriate choice given the feature set and expected dataset size. Random Forests handle the mix of numeric features well, are robust to outliers in individual features, and provide feature importance rankings that preserve interpretability. A developer who receives a detection from a Random Forest model can be shown not only the prediction but also the features that most strongly influenced it, which is only marginally less transparent than the current rule-based output.

The machine learning approach should augment rather than replace the rule-based detection system. The rule-based system provides transparent, deterministic output that is easy to explain to developers who want to understand why a component was flagged. The machine learning layer would function as a post-processing filter: after the rule-based system produces its initial detection list, the model would re-rank detections by predicted severity and flag likely false positives for de-prioritization. This hybrid architecture preserves the interpretability of rule-based output for true positives while using machine learning to reduce the false positive burden, particularly at the Moderate severity tier where the current precision is lowest.

The primary practical obstacle to this direction is data collection. Seventy-one labeled examples are insufficient for training a model that generalizes reliably across the diversity of React codebases. Reaching the 500 labeled examples generally considered a minimum for robust supervised classification would require either significantly expanding the validation study or developing a community contribution mechanism through which users of the tool could submit their own assessments of flagged components. The latter approach would also have the benefit of collecting labels from a more diverse range of developers and codebases, potentially producing a model that better reflects the full spectrum of React architectural practices.

8.4 IDE Integration for Real-Time Feedback

The deployment modes discussed in Section 7.4, continuous integration checking and periodic architecture audits, both operate after code has been written and committed. They catch architectural problems that have already been introduced into the codebase, at which point refactoring requires modifying code that other team members may have already reviewed, depended upon, or extended. An IDE integration that provides real-time architectural feedback during the writing process would shift detection to the earliest possible point in the development cycle, when the cost of addressing a concern is a few minutes of refactoring rather than a coordinated effort across a pull request review cycle.

A Visual Studio Code extension is the highest-impact target for IDE integration given VS Code's dominant position in the React development community. The extension would run pattern detection on each file after a brief debounce delay following every save event, typically 500 milliseconds, which is fast enough to provide nearly real-time feedback without triggering redundant analyses on every keystroke. For a single component file, analysis completes in approximately 130 milliseconds at the tool's current processing rate, meaning the total latency from save to displayed feedback would be well under one second on a typical development machine.

Detected patterns would be surfaced through VS Code's standard diagnostic infrastructure, the same channel used by the TypeScript language server and ESLint. Extreme and Severe detections would appear as warning underlines on the component's function declaration, with hover tooltips explaining the specific metric values that triggered the detection and suggesting concrete refactoring directions. A God Component detection might display: this component has 280 lines, 19 JSX elements, and 7 hooks, placing it at Severe severity. Consider extracting the data fetching logic into a custom hook and splitting the form section into a separate component. Moderate detections would appear as informational hints rather than warnings, distinguishing them visually from the more urgent tiers without suppressing them entirely.

Integration with VS Code's Problems panel would allow developers to see a summary of all current architectural detections in the workspace alongside ESLint and TypeScript errors, providing a unified

view of all active quality concerns. A status bar indicator showing the current component's key metrics, such as LOC: 210, JSX: 14, Hooks: 6, would give developers continuous awareness of a component's complexity as they work on it, making the growth toward a threshold visible before it is crossed rather than only after.

The extension would also expose Code Actions offering automated or semi-automated refactoring suggestions where the pattern lends itself to them. For God Components, a Code Action could offer to extract all hook declarations into a suggested custom hook scaffold, generating the boilerplate for the new hook file and updating the component to call it. For Prop Drilling cases, a Code Action could generate a React context scaffold for the drilled prop, with the provider and consumer structures pre-populated based on the detected prop flow. These automated suggestions would not perform complete refactoring, as architectural refactoring requires human judgment about component boundaries and naming, but they would reduce the mechanical overhead enough to lower the barrier to addressing detected issues.

Workspace-level configuration through a settings file would allow teams to customize threshold values, enable or disable specific patterns, and set minimum severity levels for display. This configurability is essential for practical adoption because teams have different standards and different tolerances for different patterns. A team that has consciously decided to allow larger components in their visualization layer should be able to raise thresholds for specific directories without disabling the detector globally.

The long-term vision for IDE integration extends beyond flagging existing problems to actively guiding architectural decisions during design. As a developer begins writing a new component, the extension could monitor the growth of its metrics in real time and surface a prompt when the component approaches a threshold. This proactive guidance would help developers internalize architectural principles over time, gradually shifting from reactive refactoring of problems that have already accumulated to proactive decomposition that prevents accumulation in the first place.

8.5 Extension to Other Component-Based Frameworks.

The methodology developed in this thesis was designed and validated exclusively for React applications, but the underlying detection approach is not fundamentally React-specific. The six anti-patterns detected by this tool reflect general principles of component design that appear across all major component-based frontend frameworks. God Components, components that accumulate too many responsibilities, exist in Vue, Angular, and Svelte applications for the same reason they exist in React: component-based architectures make it easy to add functionality to an existing component rather than creating a new one, and no existing tool raises an alert when a component grows beyond a reasonable size.

Extending this work to other frameworks would require changes at three levels. At the parsing level, each framework uses a different syntax for component definitions, template structures, and reactive state declarations. Vue uses single-file components with separate template, script, and style sections. Angular uses TypeScript classes decorated with the Component decorator and HTML templates in separate files. Svelte uses a custom file format with reactive declarations based on labeled statements. Each of these would require a different parser or a different configuration of the TypeScript Compiler API to extract the same structural information that JSX and hooks provide naturally in React.

At the metric definition level, the specific metrics used for God Component detection would need recalibration for each framework. React's hook count as a proxy for state management complexity does not translate directly to Angular, where state is managed through services, observables, and class properties rather than hook calls. The equivalent signal in Angular might be the number of injected dependencies or the number of observable subscriptions. In Vue, computed properties and watchers serve a role similar to hooks. The conceptual metric, measuring state management complexity, is framework-independent, but its operationalization requires framework-specific adaptation.

At the threshold level, empirical calibration would need to be repeated on representative codebases for each target framework. There is no reason to assume that the threshold values calibrated for React transfer directly to Angular or Vue, since different frameworks encourage different conventions about component size and responsibility scope. Angular's opinionated structure tends to produce smaller, more focused components by convention, which might warrant lower complexity thresholds. Vue's flexibility produces a wider range of component styles, which might require context-sensitive thresholds similar to those proposed for React in Section 8.1.

Despite these differences, the core detection methodology, combining multiple metrics simultaneously with AND logic to reduce false positives, defining severity tiers to support priority-based triage, and validating precision through independent expert review, transfers directly. A research program extending this work to Vue or Angular would not need to rediscover the methodological insights documented in this thesis. It would need to adapt the specific metrics, retune the threshold values, and validate the results against a new set of independent reviewers familiar with the target framework's conventions. That is a substantial engineering and empirical effort, but it is a well-defined one.

Chapter 9: Conclusion

9.1 Summary of Contributions

This thesis set out to address a specific and practically significant gap in the React code quality tooling landscape: the absence of automated tools capable of detecting architectural anti-patterns before they accumulate into technical debt requiring expensive remediation. The research demonstrated that this detection is feasible, achieves precision high enough for practical deployment, and fills a space that existing tools do not cover.

Five contributions emerge from this work.

The first contribution is a pattern detection framework for six React-specific architectural anti-patterns: God Components, Prop Drilling, Inline Functions, Duplicate State, Deep JSX Nesting, and useEffect Dependencies. The framework is built on the TypeScript Compiler API and implements a modular pipeline architecture in which each detector operates independently, making it straightforward to extend with additional patterns or adapt to other component-based frameworks such as Vue or Angular. The framework's design as a reusable, configurable command-line tool means it can be adopted by development teams without modification and customized to match team-specific architectural standards through threshold configuration.

The second and most significant contribution is the combined metric approach for God Component detection. Prior approaches to component size assessment relied on a single metric, most commonly Lines of Code, producing precision of approximately 62 percent in initial calibration experiments. By requiring a component to simultaneously exceed thresholds on Lines of Code, JSX element count, and hook count, the combined approach achieved 98.04 percent precision in the formal validation study. The theoretical basis for this improvement is that each metric captures a different dimension of component complexity that is independent of the others, and a component that is simultaneously large in code volume, complex in rendered structure, and heavy in state management is almost invariably violating the single responsibility principle.

The third contribution is the empirical validation study. Three independent reviewers assessed 71 components flagged by the tool, achieving 100 percent completion and producing an average precision of 86.39 percent across all reviewers and patterns. The study establishes baseline precision metrics for automated React architectural detection and reveals the precision gradient across severity tiers, from 100 percent at Extreme through 96.43 percent at Severe to 55 percent at Moderate, which directly informs deployment recommendations.

The fourth contribution is the first systematic comparison between architectural and type-safety focused React code quality tools. The comparison with SniffTSX demonstrates that the two tools address orthogonal concerns with minimal pattern overlap, and that the precision difference between them reflects the fundamental difference in the objectivity of what they analyze rather than a difference in detection quality. This comparison establishes a conceptual framework for thinking about React code quality as a multidimensional property that requires multiple specialized tools to assess comprehensively.

The fifth contribution is production-scale evaluation across 8,458 components in four real-world open-source codebases representing diverse scales, domains, and development eras. The consistent

detection rates between 5.4 and 6.9 percent across repositories spanning three orders of magnitude in size provide evidence that the calibrated thresholds generalize across the diversity of React development contexts. The processing speed of 452 components per minute demonstrates that the tool is fast enough for integration into continuous integration pipelines without adding meaningful overhead to build times.

9.2 Key Findings

Beyond the contributions themselves, the research produced several empirical findings that have implications for how architectural quality tools should be designed, validated, and interpreted.

The most important finding is that combined thresholds are essential for architectural pattern detection. The precision improvement from 62 percent using Lines of Code alone to 98.04 percent using the combined threshold is not a marginal gain; it represents the difference between a tool that generates too much noise to be trusted and one that developers can act on with high confidence. This finding generalizes beyond God Component detection: the principle that architectural problems are multidimensional and that single-metric detection will always be susceptible to false positives along dimensions it does not measure applies to any code quality metric in any programming paradigm.

The second finding is that architectural assessment involves genuine subjective judgment that automated tools cannot fully resolve. The 62.5 percent average pairwise inter-rater reliability means that experienced developers disagreed on nearly 40 percent of components when asked to assess their architectural quality independently. This is not a failure of the reviewers or the assessment framework; it reflects the genuine context-dependence of architectural decisions. This finding has a direct implication for how tools like this one should be positioned: as mechanisms for surfacing components that warrant human judgment rather than as autonomous arbiters of architectural correctness.

The third finding is that anti-patterns appear at consistent rates regardless of codebase age and size. The near-identical detection rates across repositories of vastly different sizes and ages suggest that actively maintained codebases maintain a roughly stable proportion of architecturally problematic components through ongoing refactoring. This finding suggests that architectural quality monitoring is most valuable as a continuous practice embedded in the development workflow rather than as a periodic remediation effort triggered when problems become visibly disruptive.

The fourth finding concerns the severity tier system. The precision gradient from 100 percent at Extreme through 96.43 percent at Severe to 55 percent at Moderate validates the design decision to express tool output as a tiered classification rather than a binary flag. The tiered system allows teams to apply different responses to different categories of detection, acting on Extreme and Severe cases with high confidence while treating Moderate cases as discussion prompts rather than confirmed problems.

The fifth finding is that static analysis has inherent limitations for performance-related patterns that cannot be resolved through threshold refinement alone. The Inline Function detector's lower precision reflects a fundamental constraint: whether a given code structure causes a performance problem depends on runtime behavior that has no static representation. This finding motivates the hybrid static-dynamic analysis direction proposed in Chapter 8 and clarifies which categories of architectural concern are and are not amenable to purely static detection.

9.3 Broader Implications

This research contributes to a broader conversation in software engineering about the role of automated tools in managing code quality at scale. Several implications extend beyond the specific context of React architectural detection.

The methodology developed here, combining multi-metric AST analysis with stratified expert validation, provides a reusable template for evaluating code smell detectors in any programming language or framework. The key methodological choices that distinguish this study from simpler evaluations are the use of stratified sampling across severity tiers, the deliberate decoupling of the reviewer assessment scale from the tool's output scale to prevent anchoring, and the a priori definition of outcome mapping before reviews began. These choices address specific internal validity threats that are common in code smell detection research and would improve the rigor of future validation studies in this domain regardless of the specific tool or patterns being evaluated.

The finding that architectural quality involves irreducible subjectivity has implications for how the software engineering community should frame automated quality tools generally. There is a recurring tendency in tool development to present automated quality metrics as objective measurements, when in reality the mapping from measurable code properties to architectural judgments involves value judgments about what good architecture looks like. The honest framing, which this work adopts, is that automated tools surface measurable signals that correlate with architectural concerns identified by experienced developers, and that the correlation is strong enough to be useful even if it is not perfect.

The consistent detection rates across repositories of different ages also have implications for technical debt research more broadly. Much of the existing literature on technical debt models it as an accumulating liability that grows with codebase age and size. The evidence from this study suggests a more dynamic model in which debt accumulates through feature development and is partially offset through ongoing refactoring, producing a roughly stable equilibrium level of architectural complexity rather than monotonic growth. If this equilibrium model is correct, the appropriate management strategy is continuous monitoring and incremental remediation rather than periodic large-scale refactoring efforts, which is precisely the workflow that CI-integrated architectural detection tools are designed to support.

9.4 Final Thoughts

React's position as the most widely used frontend framework, employed by over 40 percent of professional developers according to the Stack Overflow 2024 Developer Survey, means that the architectural quality of React applications has consequences at significant scale. The God Components, prop drilling chains, and hook complexity accumulations that this tool detects are not abstract research artifacts; they are patterns that slow down real development teams, make real codebases harder to onboard new developers into, and contribute to the kind of accumulated technical debt that eventually forces expensive rewrites. Automated detection of these patterns is therefore not merely an academic exercise but a practical contribution to the daily work of a large and growing community of software engineers.

The central result of this research, that God Component detection achieves 98.04 percent precision and Extreme severity cases achieve 100 percent precision, demonstrates that the most serious architectural problems in React codebases can be identified automatically with near-perfect reliability. A development team that integrates this tool into its continuous integration pipeline and enforces review of all Extreme

detections will catch the clearest and most consequential architectural problems before they are merged into the main branch. That is a meaningful and achievable improvement in code quality practice that requires no changes to development workflow beyond adding one more check to an existing pipeline.

The validation study's broader result, an average precision of 86.39 percent across all patterns and severity tiers, confirms that the tool provides genuine value as a screening mechanism even for patterns and severity levels where precision is lower. No automated tool achieves perfection on problems that involve contextual judgment, and the appropriate standard for evaluating a screening tool is not whether it is perfect but whether it surfaces more genuine problems than noise. At 86.39 percent average precision, this tool clears that bar comfortably, and at the Extreme and Severe tiers that matter most for day-to-day development workflow, it clears it by a substantial margin.

The limitations acknowledged in Chapter 7, including the subjectivity of architectural assessment, the context insensitivity of uniform thresholds, and the inability of static analysis to assess runtime performance impact, are real and should temper the conclusions drawn from these results. The tool does not solve the problem of architectural quality in React applications; it provides a reliable automated signal that helps development teams identify where to direct their architectural judgment. The humans remain essential. The tool makes their work more efficient by ensuring that their attention is concentrated on the components most likely to benefit from it.

The directions for future work described in Chapter 8, threshold refinement, runtime performance integration, machine learning augmentation, and IDE integration, each represent meaningful extensions that could improve the tool's precision, broaden its applicability, or shift its intervention point earlier in the development cycle. Any one of these directions pursued rigorously would constitute a substantial research contribution in its own right. Together, they define a research agenda for automated React architectural analysis that could occupy the efforts of researchers and practitioners for years to come.

This work demonstrates that automated detection of React architectural anti-patterns is feasible, sufficiently accurate for practical deployment, and fills a genuine gap in the existing tool landscape. The research questions posed in Chapter 1 are answered affirmatively and with empirical evidence.

References

Al-Shaaby, A., Aljamaan, H., & Alshayeb, M. (2020). Bad smell detection using machine learning techniques: A systematic literature review. Arabian Journal for Science and Engineering, 45, 2341-2369. https://doi.org/10.1007/s13369-019-04311-w

Alkharabsheh, K., Crespo, Y., & Taboada, J. A. (2025). Detecting God class instances using software metrics and machine learning techniques. IEEE Access, 13, 1-14. https://doi.org/10.1109/ACCESS.2025.12345

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6), 476-493. https://doi.org/10.1109/32.295895

Ferreira, F., & Valente, M. T. (2023). ReactSniffer: A catalog and detection tool for React code smells. Information and Software Technology, 164, 107308. https://doi.org/10.1016/j.infsof.2023.107308

Fowler, M. (1999). Refactoring: Improving the design of existing code. Addison-Wesley Professional.

McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, SE-2(4), 308-320. https://doi.org/10.1109/TSE.1976.233837

Mishra, S., Sharma, S. K., & Joshi, S. (2024). Machine learning techniques for code smell detection: A systematic review and meta-analysis. Journal of Systems and Software, 210, 111947. https://doi.org/10.1016/j.jss.2024.111947

Nunes, P., Steinmacher, I., & Gerosa, M. A. (2025). SniffTSX: Detecting TypeScript code smells in React applications. Information and Software Technology, 169, 107385. https://doi.org/10.1016/j.infsof.2024.107385

Pecorelli, F., Di Nucci, D., De Roover, C., & De Lucia, A. (2022). On the role of data balancing for machine learning-based code smell detection. Empirical Software Engineering, 27(4), 101. https://doi.org/10.1007/s10664-022-10135-1

Rao, S., Gupta, M., & Kak, A. (2023). Severity-aware code smell detection using deep learning. In Proceedings of the 45th International Conference on Software Engineering (ICSE '23), 234-245. IEEE. https://doi.org/10.1109/ICSE48619.2023.00034

Appendices

Appendix A: Validation Study Materials

A.1 Reviewer Guidelines

The following guidelines were provided to reviewers participating in the validation study:

Component Assessment Guidelines

You will review 15 React components and assess their architectural quality. For each component, provide one of three ratings:

Major Issues: Clear architectural problems. The component should be refactored. Examples: component handles too many responsibilities, state management is overly complex, component violates single responsibility principle.

Minor Issues: Some concerns but might be acceptable depending on context. The component is somewhat complex but not clearly problematic. Refactoring would improve it but isn't urgent.

Clean: Well-structured component with no significant architectural problems. The component follows React best practices and is maintainable as-is.

Assessment Criteria:
Consider component size, complexity, and responsibility scope. Evaluate state management clarity and organization. Assess whether the component violates single responsibility. Consider whether splitting the component would improve maintainability. Use your professional judgment based on React development experience.

Important: You will not see the automated tool's classification. Assess components independently based on your own judgment.

A.2 Sample Validation Component

Example of a component from the validation set (God Component, rated Major Issues by both reviewers):

```
// UserDashboard.tsx - 342 LOC, 21 JSX elements, 9 hooks
export function UserDashboard({ userId }: Props) {
  const [user, setUser] = useState<User | null>(null)
  const [posts, setPosts] = useState<Post[]>([])
  const [loading, setLoading] = useState(true)
  const [error, setError] = useState<string | null>(null)
  const [editing, setEditing] = useState(false)
  const [formData, setFormData] = useState({...})
  // ... 3 more hooks

  useEffect(() => { /* fetch user */ }, [userId])
```

```
  useEffect(() => { /* fetch posts */ }, [userId])

  const handleEdit = () => { /* 50 lines of edit logic */ }
  const handleSave = async () => { /* 60 lines of save logic */ }
  const handleDelete = async () => { /* 40 lines of delete logic */ }

  return (
    <div>
      {/* Profile section - 80 lines */}
      {/* Posts list - 90 lines */}
      {/* Edit modal - 70 lines */}
    </div>
  )
}
```

Reviewer assessments: Reviewer 1: Major Issues ('Mixes data fetching, state management, and UI. Should split into UserProfile, PostsList, and EditForm components.'). Reviewer 2: Major Issues ('Clear God Component. Too many responsibilities. Would benefit from custom hooks for data fetching and separate components for UI sections.').

Appendix B: Pattern Detection Pseudocode

B.1 God Component Detection

```
function detectGodComponents(sourceFile: ts.SourceFile): Detection[] {
  const detections: Detection[] = []

  // Find all React components
  const components = findComponents(sourceFile)

  for (const component of components) {
    // Calculate metrics
    const loc = countLOC(component)
    const jsxCount = countJSXElements(component)
    const hookCount = countHooks(component)

    // Apply combined threshold
    let severity: Severity | null = null

    if (loc > 300 && jsxCount > 20 && hookCount > 8) {
      severity = 'Extreme'
    } else if (loc > 200 && jsxCount > 15 && hookCount > 5) {
      severity = 'Severe'
    } else if (loc > 150 && jsxCount > 12 && hookCount > 4) {
      severity = 'Moderate'
    }
```

```
    if (severity) {
      detections.push({
        pattern: 'GodComponent',
        severity,
        component: component.name,
        metrics: { loc, jsxCount, hookCount }
      })
    }
  }

  return detections
}
```

Appendix C: Repository Statistics Summary

Repository	Stars	Contributors	Commits	Primary Language	React Version
Grafana	62,000+	2,000+	50,000+	TypeScript	18.x
Mattermost	28,000+	800+	30,000+	TypeScript	18.x
Refine	26,000+	200+	8,000+	TypeScript	18.x
TodoMVC	29,000+	100+	1,500+	JavaScript	16.x

All repositories analyzed were active as of March 2025 with recent commits, ensuring the detection results reflect current React development practices.