Agent-Oriented Linting for Generated Cross-Platform Applications

Daniel Chen
Anything

Arnav Surve
Anything

Abstract

Large language model agents increasingly generate complete application code for web, mobile, and backend targets, but generated applications fail in ways that traditional linters do not emphasize. In particular, generated React, Next.js, Expo, React Native, and serverless TypeScript projects often fail through framework-specific integration mistakes: browser APIs used during server rendering, missing declarations that a component must run on the client, invalid routing conventions, mobile layout constraints, unsafe serverless loops, or platform-incompatible imports. These defects may be syntactically valid, type-correct, and invisible until build, preview deployment, device testing, or runtime.

We present *laint*, an expert-curated benchmark for generated JSX and TSX applications, implemented as an agent-oriented linting system. Laint encodes platform-specific failure modes as lightweight static analyses over Babel abstract syntax trees (ASTs) and exposes them through a programmatic API, command-line interface, and agent hook. Because the hook runs immediately after file edits, it is intended to shorten the agent repair cycle by surfacing framework-specific problems before slower build, preview, device, or runtime checks. The current benchmark contains 55 rules across Expo/mobile, web, and backend targets, with generated rule metadata and documentation. We describe the design principles behind the rule corpus and a protocol for measuring model behavior on generated application traces: prompts, generated files, lint findings, build results, runtime logs, and diagnostic-feedback repair attempts. Our central claim is that framework-aware lint rules selected by human experts can serve as a benchmark for comparing language models on generated-app correctness and coding habits, while also acting as repair signals that measure whether coding agents comply with localized static feedback.

1 Introduction

AI agents can now produce multi-file applications rather than isolated snippets. This changes the role of static analysis. A conventional linter is primarily a tool for human developers working in a stable codebase. An agent-facing linter also acts as feedback in a generate-check-repair loop, where a model writes code, receives diagnostic output, and edits the code again. To be useful in that loop, the linter must detect likely failures early, report them in terms the agent can act on, and avoid noisy findings that derail useful work.

Generated web and mobile applications expose a recurring class of defects that sit between syntax, type checking, and framework runtime behavior. For example, a Next.js component can be valid TypeScript while still reading `window` during server-side rendering. An Expo Router screen can be syntactically correct while using relative navigation paths that resolve incorrectly. A serverless function can pass local tests while containing an unbounded loop that is likely to time out in production. These are not merely style preferences; they are repeated integration failures in generated applications.

Laint was built to target this gap. It is a compact lint-rules package for JSX and TSX code, designed to run after an agent edits a file and return concrete findings that the agent can repair

immediately. More importantly for this paper, the expert-selected rule set is itself the benchmark: given a fixed prompt suite run across the same grid of models, laint measures which models produce framework-specific defect patterns and which rule-defined classes they trigger. A second measurement is diagnostic compliance: after a model receives rule-specific feedback, does it follow that prompt and revise the code toward the benchmark constraints? The remaining human labeling task is not to decide whether these defect classes are bad; that has already been encoded in the benchmark. Instead, labeling determines whether each model output truly contains the defect reported by laint. These findings are useful not only as negative scores, but also as behavioral observations about model tendencies: for example, a model that frequently emits emoji characters as icons is revealing a distinct interface pattern learned from training examples, even when that pattern is undesirable for a production app. Rather than trying to replace ESLint [7], TypeScript [6], or framework compilers, laint focuses on rules that are specific, cheap to run, and operationally tied to known generated-app failures.

This paper makes three contributions:

1. We define *agent-oriented linting*: static analysis designed for code generation feedback loops rather than only human code review.
2. We describe the design of laint, a rule-based JSX/TSX benchmark with platform-tagged checks for Expo/mobile, web, and backend generated applications.
3. We present a rule taxonomy and benchmark protocol for measuring one-shot model behavior and diagnostic compliance on generated web, mobile, and backend application code.

2 Motivation

Generated applications fail in ways that reflect both the target framework and the generator's learned habits. In internal use, many defects were not exotic compiler problems. They were small but consequential choices: using a browser API in a server-rendered module, importing React Native primitives into a web project, omitting a `response.ok` check, using unsupported animation patterns, or forgetting an Expo-specific layout guard. These problems are easy to fix once identified, but expensive when discovered only after preview, deployment, or user interaction.

Traditional lint rules can catch some of these issues, especially generic JavaScript and React anti-patterns. The gap is that generated applications combine multiple fast-moving platform contracts: rules imposed by frameworks, deployment targets, and device runtimes that generated code must obey. Mobile contracts are especially easy to miss because a browser-only preview does not exercise native layout, navigation, or device-input behavior. In principle, many of these checks could be implemented as custom ESLint rules. Laint's narrower claim is operational: it packages generated-app-specific checks as a small, platform-profiled tool that can run immediately on edited files and return structured repair feedback without requiring a full project lint configuration or a slower build. Agent-oriented linting treats such specificity as a feature: if a rule captures a recurring generated-app defect and produces a reliable repair signal, it is worth encoding.

3 System Design

Laint parses JSX and TSX source with Babel [1], runs selected rule functions over the abstract syntax tree (AST), and returns structured results containing the rule name, message, source location, and severity. Rules can be selected explicitly, excluded from an all-rules run, or chosen by platform.

Platform mode runs rules tagged for a target such as `expo`, `web`, or `backend`, plus universal rules that apply regardless of platform.

The system is intentionally small. Each rule exports both an analysis function and metadata: name, severity, platform tags, category, and description. A synchronization script generates the central registry and README rule table from those per-rule metadata exports. This avoids a common failure mode in rule-heavy projects where implementation, registry, tests, and documentation drift apart.

Laint exposes three interfaces:

- a programmatic API for embedding lint checks in other tools;
- a command-line interface (CLI) for checking individual files; and
- an agent hook, a small integration point that runs after file edits and feeds findings back to the coding agent.

The hook interface is important because it moves feedback closer to the model action that introduced the defect. Instead of waiting for a later lint, build, preview, or runtime step, the agent receives the finding immediately after the edit and can repair while the local context is still fresh. A finding is not merely a report; it becomes a prompt for the next repair action.

4 Rule Taxonomy

The current laint implementation contains 55 rules and 59 test files. Table 1 summarizes the rule corpus by category. These categories are taken from the `category` field in each rule's metadata rather than assigned after the fact for the paper. The corpus contains 15 error-level rules and 40 warning-level rules. Seventeen rules are universal, while the remaining rules target Expo, web, backend, or a combination of platforms.

Category	Rules
Code Style	14
React Native / Expo	9
React / JSX	6
Next.js	5
Backend / SQL	5
Screen Transitions	5
Liquid Glass	4
Expo Router	2
Tailwind CSS	2
Error Handling	1
General	1
URL	1

Table 1: Rule categories in the current laint implementation.

Version pinning. All rule counts and reported benchmark artifacts in this paper are tied to a fixed repository state: `main` commit `6a60a0295955ee6cc1d639c88955ea50722e3516`, dated 2026-05-14. The counts are reproducible from checked-in repository artifacts using the `paper:stats` script documented with the paper source. The expanded-grid and repair-loop artifacts also include metadata for the runner, prompt IDs, model aliases, model IDs, and token or repair-turn limits. This pin is primarily for citation and follow-up work: later papers or benchmark updates should cite either an immutable commit hash or a purpose-named git tag so that later rule additions, rule rewrites, or prompt-suite changes do not change the meaning of previously reported results.

The categories reflect several distinct failure modes. The "Liquid Glass" category refers to rules for Expo glass-effect components, such as requiring availability fallbacks and avoiding style combinations that break the effect.

Rendering and hydration. Rendering rules target mismatches between React [5] code that appears valid locally and framework constraints imposed by server rendering or client-component boundaries. In frameworks such as Next.js, some components render first on the server and later hydrate in the browser; code that reads browser-only globals too early can fail in this transition. Examples include guards for browser-only APIs, checks that browser APIs move into effects, required client-component directives, and checks against server-only imports from client files.

Platform compatibility and mobile UI constraints. Mobile and platform-compatibility rules prevent generated code from mixing incompatible APIs or violating layout constraints. Examples include checks for web/native import boundaries, Expo image imports, safe-area handling around notches and home indicators, keyboard avoidance around text inputs, and bottom padding for native tab screens. These are common in agent-written code because examples for web and native React are semantically similar but operationally distinct.

Framework conventions. Expo [3], Next.js [9], Tailwind, and screen-transition rules encode conventions that are not always enforced by the compiler. Examples include absolute route paths, tab header configuration, animation worklet directives, transition progress ranges, shared-transition tag matching, and animation class restrictions. These are not arbitrary style preferences; they are small framework contracts that generated code often violates while still remaining valid TypeScript.

Encoding these contracts as rules also changes the token economics of the repair loop. An agent could search documentation or retrieve framework examples after every failure, but that is nondeterministic, token-intensive, and often too broad for the local edit. A laint diagnostic compresses the relevant convention into a deterministic, file-local signal that can be fed back to the model directly.

Runtime robustness. Backend and error-handling rules target defects that often pass static type checks: missing `response.ok` checks, synchronous filesystem calls, nested SQL template calls, unrestricted loops in serverless functions with execution time limits, missing structured error fields, and unsafe JSON parsing.

5 Agent-Oriented Rule Design

Laint rules are designed around repairability. A good agent-oriented rule should satisfy four criteria.

Specificity. The rule should identify a narrow failure mode rather than a broad aesthetic preference. Specificity reduces false positives and makes the suggested repair obvious.

Locality. The rule should usually be decidable from the edited file. This keeps checks fast enough to run after every agent edit and avoids requiring a full index of every import, route, type, and configuration file in the project.

Operational grounding. The rule should correspond to observed build, preview, runtime, or user-experience failures in generated projects. This differs from style rules whose value is primarily consistency.

Actionable output. The result should be phrased so an agent can repair it directly. For example, "Files using client-only features must have a `"use client"` directive" is more useful in an edit loop than a generic server-rendering warning.

6 Benchmark Protocol

The simplest useful benchmark is a prompt-to-code study. Given a suite of realistic app-building prompts, ask one or more language models to produce JSX/TSX files for web, mobile, and backend scenarios. Run laint over the generated files and count reported findings by model, rule, and platform. This produces a raw, rule-defined signal about model behavior on framework-aware generated-app correctness: which expert-selected defect patterns does a model appear to produce, and how often? The same counts also describe qualitative model tendencies, such as whether a model prefers inline styles, loose type assertions, silent error handling, emoji icons, or web-centric APIs in mobile code.

A first-pass study can be intentionally lightweight. For each prompt-model pair, record the generated code, laint findings, and enabled platform profile. Because the benchmark rules are already expert-selected failure modes, the primary raw signal is the number and distribution of reported findings a model produces. A follow-up labeling pass estimates detector quality. Precision can be estimated by labeling reported findings:

$$\text{precision} = \frac{\text{valid findings}}{\text{valid findings} + \text{invalid findings}}$$

Recall requires a different denominator: all true instances of the expert-defined defect classes, including instances laint did not report. Estimating recall therefore requires an independent review of generated files for missed defects, a seeded-defect corpus, or another oracle that can identify true instances beyond laint's own output. When that denominator is available, recall is:

$$\text{recall} = \frac{\text{valid findings}}{\text{valid findings} + \text{missed defects}}$$

Precision and recall can be combined with an F-score when a single detector-quality number is useful. The balanced version, F1, weights precision and recall equally:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

A benchmark can also report F_β when one side matters more. Values of $\beta > 1$ weight recall more heavily, while values of $\beta < 1$ weight precision more heavily:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}$$

Here, "valid" means that the generated code actually contains the expert-defined defect reported by the rule. A "missed" defect is a true instance of the same defect class that laint failed to report. These labels do not mean that a human is re-deciding whether the rule describes a bad pattern. Ambiguous findings and ambiguous missed cases should be reported separately rather than folded into either side. The labeled findings can also be grouped by rule and category to identify high-confidence rules, noisy detectors, and platform-specific gaps.

7 Pilot Evaluation

Preliminary grid. As an initial small-scale evaluation, we ran six prompts across three models: `openai-gpt-5.5`, `anthropic-sonnet-4.6`, and `google-3.1-pro`. The prompt set covered two web tasks, two Expo/mobile tasks, and two backend tasks. All 18 generations completed and parsed successfully. Laint reported 240 raw findings, producing a labeling queue for estimating detector precision on these model outputs and a starting point for recall-oriented review. Table 2 summarizes the run.

Metric	Value
Prompts	6
Model aliases	3
Grid slots	18
Completed generations	18
Parse errors	0
Generation errors	0
Reported findings	240

Table 2: Preliminary prompt-to-code benchmark run before detector-quality labeling.

Expanded grid. We then ran a larger raw grid over the same six prompts and seven configured model aliases. The run artifact is included with the paper source, and the following tables are generated from it by `npm run paper:tables`. Table 3 gives the overall run shape; Tables 4–7 break the same run down by model, prompt, rule, and prompt-model pair.

The expanded tables make the raw benchmark results easier to inspect than a single aggregate count. First, findings are concentrated: the twelve most frequent rules account for 422 of 476 findings, or 88.7% of the run. The largest buckets are not obscure compiler failures; they are recurring generated-code habits such as inline styles, silent branches, type assertions, emoji icons, and optional prop shapes. Second, model behavior differs substantially even before detector-quality labeling. The two Anthropic aliases produce the highest raw finding counts per linted run, while the Google aliases produce fewer reported findings but also include one parse failure for `google-2.5-flash`. That failure should be read partly as an output-budget/truncation case: the recorded finish reason was `length`, with 11,996 completion tokens. Third, task shape matters: the two web prompts produce the largest prompt-level finding totals, while the backend prompts have fewer laint findings but still expose parse and reliability failures.

Metric	Value
Prompts	6
Model aliases	7
Grid slots	42
Completed generations	36
Parse errors	2
Generation errors	6
Reported findings	476

Table 3: Expanded raw prompt-to-code benchmark run before detector-quality labeling.

Model alias	Linted runs	Parse errors	Gen. errors	Findings	Findings/linted
`anthropic-sonnet-4.6`	6	0	0	127	21.2
`anthropic-opus-4.6`	6	0	0	123	20.5
`openai-gpt-5.5`	6	0	0	78	13.0
`openai-gpt-5.4`	5	1	0	59	11.8
`google-3.1-pro`	6	0	0	47	7.8
`google-2.5-flash`	5	1	0	42	8.4
`moonshot-kimi-k2.6`	0	0	6	0	−−

Table 4: Expanded-grid findings by model. Linted runs exclude generation failures and parse failures.

These counts are raw benchmark signals because the rules encode expert-selected defect classes, but they are not yet validated defect rates. The next step is to label findings as valid, invalid, or ambiguous to estimate detector precision and report rule-level noise. A recall estimate additionally requires reviewing generated files for true defect instances that laint did not report; once both precision and recall are available, the same labels can produce F1 or another F-score. The configured `moonshot-kimi-k2.6` alias failed all six generations because of provider authentication or network infrastructure failures, so it is retained in the artifact as an infrastructure failure but excluded from model-quality comparisons. A scored Kimi comparison requires rerunning the grid after the credential path is fixed.

Diagnostic-compliance pilot. We also ran a repair-loop pilot on the same expanded-grid outputs. For each completed baseline generation, we fed the current code and laint diagnostics back to the same model and asked for a revised complete file. We repeated this for up to three turns, stopping early when the file had zero laint findings and no parse error. We interpret this as a diagnostic-compliance measurement: the model is being tested on whether it can follow localized static feedback, not merely whether it can generate a plausible file from the original task prompt. This pilot does not replay the exact CLI hook or a full autonomous coding-agent loop; it tests the same diagnostic content in a direct repair prompt. Table 8 summarizes the repair loop; Tables 9 and 10 break the same run down by model and prompt.

The repair loop reduced raw reported findings from 476 to 101, a net reduction of 375 findings, or 78.8%, within three turns. At the rule level, 445 findings disappeared and 70 new findings were introduced during repair, showing why the net count should not be read as literal fixed defects. It also reduced parse errors from two to one; the remaining final parse error again had finish reason `length`. Eighteen of the 36 repairable records reached zero findings and no parse error, and seven reached that state after a single repair turn. The result is not uniformly positive: web and backend

Prompt	Platform	Linted runs	Parse errors	Gen. errors	Findings
`taskflow-web`	web	6	0	1	125
`chat-web`	web	5	1	1	92
`insurance-reports-backend`	backend	6	0	1	84
`event-planner-mobile`	expo	6	0	1	77
`beauty-shop-mobile`	expo	6	0	1	61
`wallet-api-backend`	backend	5	1	1	37

Table 5: Expanded-grid findings by prompt and target platform.

Rule	Category	Findings	Share
`no-inline-styles`	Tailwind CSS	102	21.4%
`no-silent-skip`	Code Style	87	18.3%
`no-type-assertion`	Code Style	64	13.4%
`no-emoji-icons`	Code Style	37	7.8%
`no-optional-props`	Code Style	29	6.1%
`scrollview-horizontal-flexgrow`	React Native / Expo	25	5.3%
`prefer-named-params`	Code Style	17	3.6%
`no-safeareaview`	React Native / Expo	14	2.9%
`browser-api-in-useeffect`	React / JSX	12	2.5%
`no-stylesheet-create`	React Native / Expo	12	2.5%
`textinput-keyboard-avoiding`	React Native / Expo	12	2.5%
`catch-must-log-to-sentry`	Error Handling	11	2.3%
Other rules	–	54	11.3%

Table 6: Most frequent expanded-grid reported findings by rule. The top twelve rules account for most raw findings.

prompts repaired more reliably than the Expo/mobile prompts, and `event-planner-mobile` barely improved overall. This makes diagnostic compliance a useful second benchmark axis: the one-shot score measures which rule-defined findings a model produces, while the repair-loop score measures how well the same model can comply with localized static feedback without introducing new reported failures.

Prompt	GPT-5.5	GPT-5.4	Sonnet 4.6	Opus 4.6	G-3.1-Pro	G-2.5-Flash	Kimi K2.6
`taskflow-web`	14	7	38	44	11	11	G
`chat-web`	14	17	37	11	13	P	G
`event-planner-mobile`	11	18	16	17	7	8	G
`beauty-shop-mobile`	11	10	8	16	7	9	G
`wallet-api-backend`	10	P	14	5	4	4	G
`insurance-reports-backend`	18	7	14	30	5	10	G

Table 7: Run-level expanded-grid finding counts. P denotes a generated file that failed parsing; G denotes a generation failure.

Metric	Value
Baseline records	42
Skipped baseline generation errors	6
Attempted repairs	36
Maximum repair turns	3
Baseline reported findings	476
Final reported findings	101
Net finding reduction	375 (78.8%)
Rule-level findings resolved	445
Rule-level findings introduced	70
Baseline parse errors	2
Final parse errors	1
Clean after one turn	7
Clean after max turns	18
Repair generation errors	0

Table 8: Diagnostic-compliance repair-loop pilot over the expanded grid. Each attempted repair feeds laint diagnostics back to the same model for up to three turns.

Model	Initial	Final	Net red.	New	Clean final	Avg. turns
Opus 4.6	123	11	91.1%	2	2/6	1.5
GPT-5.5	78	8	89.7%	8	4/6	1.3
Sonnet 4.6	127	63	50.4%	55	2/6	3.0
GPT-5.4	59	6	89.8%	0	4/6	2.0
G-3.1-Pro	47	5	89.4%	2	4/6	1.5
G-2.5-Flash	42	8	81.0%	3	2/6	2.5

Table 9: Diagnostic-compliance outcomes by model, excluding baseline generation failures. Net red. is net reported-finding reduction; New counts rule-level findings introduced during repair. Average turns is computed over runs that reached zero findings and no parse error.

Prompt	Platform	Initial	Final	Net red.	New
`taskflow-web`	web	125	4	96.8%	2
`chat-web`	web	92	5	94.6%	0
`insurance-reports-backend`	backend	84	5	94.0%	1
`beauty-shop-mobile`	expo	61	13	78.7%	6
`wallet-api-backend`	backend	37	4	89.2%	2
`event-planner-mobile`	expo	77	70	9.1%	59

Table 10: Diagnostic-compliance outcomes by prompt and platform. New counts rule-level findings introduced during repair.

8 Future Benchmark Extensions

A fuller benchmark should answer four model-evaluation questions.

RQ1: Behavioral profiles. Which framework-aware coding habits does each model exhibit? Rule distributions can characterize model-specific tendencies such as overusing inline styles, inserting emoji as icons, relying on type assertions, skipping error branches silently, or mixing web and native APIs.

RQ2: Early detection. For each model, how often does laint identify defects before TypeScript, framework builds, preview deployment, or runtime interaction? This can be measured by replaying generated application traces and recording the earliest stage at which each model's defects are detected.

RQ3: Precision, recall, and F-score. For each model and rule category, what fraction of laint findings are true instances of the expert-defined defect class, and what fraction of all true instances does laint report? Precision measures noise in the agent feedback loop. Recall measures coverage of the expert-defined defect classes. F-score combines the two when a single detector-quality metric is needed. Precision can be estimated by manual labeling of reported findings; recall requires a labeled corpus that also includes missed defects, created through independent manual review, seeded examples, or runtime/build failures traced back to rule classes.

RQ4: Diagnostic compliance. When findings are fed back to the same model or agent, how often does the next edit comply with the requested correction, resolve the issue, and avoid introducing new failures? This can be measured by running the same generation tasks with and without the laint hook and comparing net finding reduction, rule-level resolved and introduced findings, turns to a lint-clean state, final build success, preview success, and number of repair iterations by model.

We propose evaluating on a corpus of generated applications from JSX/TSX app-building tasks spanning web, mobile, and backend targets. For each task, the benchmark should capture prompts, model identity, generated code, lint output, type-check results, build results, runtime logs, mobile simulator or device-preview outcomes where applicable, and final human or automated acceptance labels. The primary comparison is between models and repair loops, not between laint and ESLint as replacements. ESLint, TypeScript, framework builds, and runtime preview form the existing sequence of diagnostics against which laint's earlier or more specific signals can be compared.

9 Discussion

The main tradeoff in laint is that many rules examine a single file and use syntactic approximations rather than whole-program analysis. For example, a hydration rule can flag a browser API reference in a server-rendered module without proving every possible render path, and a mobile compatibility rule can flag an import pattern without running the app in a device simulator. This is intentional: in an agent feedback loop, the goal is not a sound proof of correctness, but a fast and concrete repair signal immediately after code is generated. A rule that catches a repeated hydration failure at edit time can be useful even if a deeper framework build would eventually report a related error. Conversely, a noisy approximation is harmful because it consumes agent iterations and may cause unnecessary code churn.

Another tradeoff is specificity. Some laint rules encode conventions that are not universal across all React or Expo projects. Platform tags and explicit configuration modes address this by letting users choose a rule set appropriate to the generated project. This is especially important for V2 filesystem projects, where TSX, Next.js server rendering, and Lambda deployment constraints differ from older JSX-oriented generated apps.

10 Limitations

The current rule corpus is shaped by failures observed in one app-generation environment, so the taxonomy may not generalize to every AI coding workflow. A mature benchmark should therefore separate rules with broad framework relevance from rules that encode Anything-specific product constraints.

The benchmark can also overfit to known failures. A held-out task set and chronological split can reduce this risk: rules should be tested on generation traces created after the rule design period, or on tasks that were not used when deciding which rules belonged in laint.

The pilot evaluation is intentionally small: six prompts, one generation per prompt-model pair, and no repeated samples or uncertainty estimates. It reports raw laint findings rather than fully labeled precision, recall, or F-scores. The current artifacts also do not measure TypeScript success, framework build success, preview behavior, mobile simulator behavior, runtime logs, or human acceptance of the generated application.

Finally, measuring diagnostic compliance is sensitive to the underlying model, prompt, provider route, output-token budget, and agent wrapper. The repair-loop pilot measures compliance against the laint benchmark itself through a direct repair prompt; it does not by itself prove that the repaired application builds, previews, or satisfies the user's intent, and it should not be treated as a full measurement of the production agent hook. A useful benchmark should report the agent configuration and pair laint-repair outcomes with build, runtime, and human acceptance labels before claiming that laint improves all coding agents equally.

11 Related Work

Laint builds on a long tradition of static analysis and linting for JavaScript and TypeScript, including ESLint [7], TypeScript [6], Babel-based transforms [1], and framework-specific lint plugins for React [5], Next.js [9], and Expo [3]. The distinctive focus is not the syntax-tree traversal itself, but the placement of linting inside an automated code-generation loop and the emphasis on generated-app failure modes across web, mobile, and backend surfaces.

The system is also related to work on language models for code and tool-augmented refinement. Codex demonstrated that large language models can synthesize code from natural-language prompts [2]. Self-Refine and Reflexion study iterative feedback loops in which generated outputs are improved using critique, execution signals, or verbal feedback [4, 8]. Laint contributes a practical instance of this feedback-loop pattern specialized for modern JSX/TSX application frameworks: instead of relying only on tests or compiler output, it supplies small, framework-aware repair signals immediately after file edits.

12 Conclusion

Agent-generated applications need feedback loops that catch framework-specific defects before they become preview or runtime failures. Laint demonstrates a lightweight approach: encode

recurring generated-app failures as repairable JSX/TSX lint rules, tag them by platform, and run them automatically after agent edits. The same expert-curated rule corpus defines a benchmark for language models: prompt models to generate applications, count reported findings, feed those findings back as repair signals, estimate detector precision, recall, and F-score on model outputs, and compare defect patterns and diagnostic compliance across models. The current 55-rule implementation suggests that many high-value checks are small, file-local, and operationally grounded. The next step is to complete detector-quality labeling for the expanded model grid and report labeled model-level benchmark results.

References

[1] Babel. Babel: The compiler for next generation javascript. https://babeljs.io/. Accessed 2026-05-16.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] Expo. Expo: React native framework and platform. https://expo.dev/. Accessed 2026-05-16.

[4] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

[5] Meta Open Source. React: The library for web and native user interfaces. https://react.dev/. Accessed 2026-05-16.

[6] Microsoft. TypeScript: Javascript with syntax for types. https://www.typescriptlang.org/. Accessed 2026-05-16.

[7] OpenJS Foundation. ESLint: Find and fix problems in your javascript code. https://eslint.org/. Accessed 2026-05-16.

[8] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

[9] Vercel. Next.js: The react framework for the web. https://nextjs.org/. Accessed 2026-05-16.