Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.080737

ARTICLE

# Constrained LLM-Guided Refactoring of JavaScript: A Smell-Targeted Transformation Framework with Human-in-the-Loop Validation

**Emir Kuanyshev and Hashim Ali***

Department of Computer Science, Nazarbayev University, Astana, Kazakhstan

*Corresponding Author: Hashim Ali. Email: hashim.ali@nu.edu.kz

Received: 13 February 2026; Accepted: 08 April 2026; Published: 08 May 2026

**ABSTRACT:** Refactoring improves maintainability without altering externally observable behavior, yet it remains costly and error-prone when applied manually at scale. While large language models (LLMs) can generate plausible refactorings, practical adoption is limited by uncontrolled edit scope, inconsistent outputs under stochastic decoding, and weak traceability of why a change was produced. This paper proposes a smell-targeted, scope-bound refactoring framework for JavaScript that couples deterministic AST-based smell detection with constrained LLM transformation. The key design principle is to bind generation to explicitly detected smell instances, enforce a structured output contract (refactored code plus per-smell rationale), and log full refactoring artifacts for repeatable evaluation. We implement the framework as a microservice-based prototype (detector, prompt construction and routing, orchestrator, analytics, and UI) and evaluate it on LeetCode-style solutions and multiple GitHub repositories. Across the evaluated projects, the approach achieves an average smell reduction of 83.96% and an average maintainability index improvement of +5.366, while maintaining a mean developer acceptance rate of 91.66%. A targeted temperature study identifies an operating point around 0.4 that maximizes acceptance (approximately 95% in controlled trials), balancing determinism with sufficient flexibility for structure-improving edits. These results suggest that explicit scope control and structured traceability are central to making LLM-based refactoring reliable and reviewable, and motivate future integration with automated validation (tests, linting) and repository-conditioned policies.

**KEYWORDS:** Large language models; automated refactoring; code smell detection; JavaScript; constrained generation; maintainability index; human-in-the-loop evaluation

## 1 Introduction

Modern software systems accumulate maintainability debt as features evolve under schedule pressure, and refactoring remains a primary mechanism for improving internal structure while preserving externally observable behavior [1,2]. In practice, however, refactoring is frequently delayed because it requires (i) reliably identifying maintainability issues (often described as code smells), (ii) selecting suitable transformations, and (iii) validating that behavior has not changed. These challenges are amplified in JavaScript ecosystems, where heterogeneous coding conventions, rapid framework churn, and diverse project structures make consistent quality enforcement difficult at scale [3,4].

Large Language Models (LLMs) have recently been explored as program transformation engines capable of generating context-aware edits and refactoring suggestions [5,6]. Empirical studies indicate that LLMs can perform targeted refactorings (e.g., Extract Method) when supported by well-designed prompts and

adequate contextual grounding [7,8]. Nonetheless, using LLMs for behavior-preserving refactoring introduces reliability risks that are particularly problematic in professional workflows: unconstrained generation may introduce edits beyond the intended scope, outputs can vary substantially with decoding settings, and the lack of structured traceability can hinder review and accountability [9,10]. As a result, the core barrier is not whether LLMs can generate readable code, but whether they can be *controlled* to produce auditable, scope-bounded transformations that developers are willing to accept.

This paper addresses that gap by proposing a smell-targeted transformation framework that couples deterministic AST-based detection with constrained LLM refactoring. The framework enforces a separation between (i) *what* should be changed (explicit smell instances reported by a detector) and (ii) *how* it should be changed (LLM-guided transformation under explicit constraints). Concretely, the system (a) binds the LLM editing scope to detected smell instances, (b) requires a structured output contract (refactored code plus per-smell rationale), and (c) persists full refactoring artifacts to enable repeatable evaluation and acceptance-aware configuration.

We study the following research questions:

**RQ1:** To what extent can a smell-targeted, scope-bound framework detect and refactor multiple JavaScript smell types within a unified workflow?

**RQ2:** Which LLM decoding configuration best balances acceptance and the risk of scope drift in behavior-preserving refactoring?

**RQ3:** How effective is the framework across heterogeneous real-world repositories in reducing smells and improving maintainability indicators?

The contributions of this work are:

1. A deterministic, extensible AST-based smell detection service for JavaScript with structured smell-instance outputs.
2. A constrained LLM refactoring protocol that binds generation to detected smell instances and enforces structured, machine-parseable outputs with per-smell rationales.
3. An end-to-end microservice architecture that logs complete refactoring artifacts (inputs, detected smells, prompts, outputs, explanations, reviewer decisions) to support traceability and repeatable analysis.
4. An empirical evaluation across controlled and real-world codebases, reporting smell reduction, maintainability index change, and developer acceptance as adoption-facing outcomes.
5. A configuration study identifying an acceptance-optimized decoding operating point, supporting practical deployment guidance.

Unlike prior approaches that focus on a narrow refactoring operation or rely on fixed external analyzers with limited smell coverage [5,11], our framework treats refactoring as a *controlled transformation problem*: detection defines the edit budget, constrained prompting defines admissible transformation behavior, and human-in-the-loop acceptance operationalizes deployability. In the evaluated repositories, the framework achieves 83.96% average smell reduction, +5.366 mean maintainability index improvement, and 91.66% mean acceptance rate, indicating that controlled LLM generation can produce refactorings that are both measurable and reviewable.

The remainder of this paper is organized as follows: Section 2 reviews related work on rule-based refactoring tools, learning-based and LLM-assisted refactoring, and explainability for developer trust; Section 3 describes the proposed framework and evaluation protocol, including metric definitions and aggregation; Section 4 reports empirical results on smell reduction, maintainability and LoC changes, acceptance

results, qualitative error modes, and user feedback; finally, Section 5 discusses implications, practical deployment considerations, limitations, threats to validity, and future directions.

## 2 Related Work

Automated refactoring research spans (i) rule-based refactoring tools, (ii) learning-based and LLM-assisted approaches, and (iii) techniques that improve trust and reviewability of automated changes. Our focus is behavior-preserving refactoring, where uncontrolled edits and weak traceability can directly limit adoption.

### *2.1 Rule-Based and Tool-Supported Refactoring*

Classical refactoring tools encode refactorings as deterministic program transformations driven by rules or patterns [12,13]. Such approaches are typically reliable and auditable but limited in coverage and adaptability: they often support a constrained set of smells/refactorings, can be brittle under diverse coding styles, and require engineering effort to extend to new patterns or languages [3,14]. These limitations motivate techniques that can generalize refactoring intent across heterogeneous projects without manually encoding every transformation rule.

### *2.2 Learning-Based and LLM-Assisted Refactoring*

Learning-based methods and LLMs have been studied as flexible engines for code generation and transformation [15,16]. LLM-assisted refactoring tools report promising results on specific refactoring tasks when guided by prompt engineering and contextual grounding [5,7,8]. However, LLM outputs remain sensitive to decoding configuration and prompt phrasing, and can introduce edits beyond the intended refactoring scope or produce changes that are difficult to audit, particularly when behavior preservation cannot be automatically verified [9,17–19]. Thus, a central open challenge is *controllability*: ensuring that LLM-driven refactoring is scope-bounded, repeatable, and traceable. Fig. 1 summarizes common LLM application categories for code changes and motivates our focus on controlled transformation for refactoring.

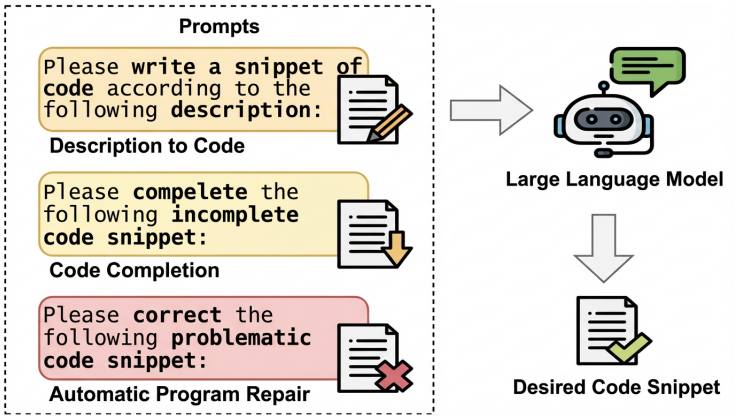


**Figure 1:** LLM application categories for code changes.

### *2.3 Explainability and Trust in Automated Code Changes*

For automated changes to be accepted in practice, developers must understand what changed, why it changed, and whether the change is safe [20,21]. Conventional tools provide structured refactoring names and diffs, while LLM-generated edits may not map cleanly onto standard refactoring operations. Without

structured rationales and artifact traceability, reviewers may reject otherwise beneficial changes due to uncertainty and perceived risk.

### *2.4 Positioning of This Work*

This paper combines deterministic smell detection with constrained LLM transformation to improve controllability and reviewability. In contrast to approaches that either (i) rely primarily on fixed refactoring rules, or (ii) use LLMs with limited scope constraints, our framework binds generation to explicitly detected smell instances, enforces a structured output contract, and logs complete refactoring artifacts and acceptance decisions. This design targets the practical bottlenecks that restrict adoption: scope drift, instability under stochastic decoding, and limited auditability in human review.

## 3 Methodology

### *3.1 Research Design and Workflow*

This study follows a design-and-evaluate methodology: (i) we implement an automated refactoring pipeline for JavaScript, (ii) operationalize refactoring as a two-stage process—detection followed by LLM-guided transformation under constraints, and (iii) evaluate the pipeline on representative code drawn from controlled and real-world sources using objective quality indicators and developer-facing acceptance outcomes.

At a conceptual level, the pipeline enforces a clear separation between what should be changed (smell detection) and how it should be changed (refactoring generation). This separation is critical for refactoring tasks because the primary risk is not "poor formatting" but unintended semantic drift (accidentally changing program behavior). The pipeline therefore constrains the LLM to act only on explicitly identified targets, and it requires the model to return both (a) refactored code and (b) a structured explanation that can be inspected during review. The end-to-end operational workflow is implemented in the following sequence:

1. *Input acquisition:* A developer submits JavaScript code to the system using the prototype web interface (or API entry point), enabling the pipeline to be applied at the file/snippet level without requiring manual refactoring setup.
2. *Smell detection:* The code smell detection service parses the code and returns a structured list of smell instances, each described by smell type, approximate location (e.g., node span/line range), and smell-specific metadata needed to guide refactoring (e.g., function name, statement count, threshold condition).
3. *Prompt construction:* The LLM service constructs a smell-conditioned prompt that includes (i) the original code, (ii) the detected smell report, (iii) explicit constraints on scope and output format, and (iv) smell-specific guidance/examples that describe acceptable refactoring patterns.
4. *Refactoring generation:* The LLM produces refactored code and a structured explanation describing what was changed and why, explicitly aligned to the detected smell instances.
5. *Review and logging:* The developer reviews the suggested refactoring in the user interface (UI) and accepts or rejects the recommendation. The system stores the original code, the smell report, the refactoring generated and the decision as a refactoring record. These records are later used to compute aggregate indicators (smell reduction, maintainability change, acceptance rate) for evaluation.

This design is intentional for three reasons. First, it makes the refactoring scope auditable by binding edits to known smell instances rather than allowing free-form rewriting. Second, it supports extensibility: new smell rules can be added to the detector without redesigning the overall pipeline. Third, it reduces the

likelihood of unintended modifications by turning the LLM into a constrained executor of smell-targeted changes rather than an unconstrained re-authoring mechanism.

### *3.2 Scope-Bounded Transformation Protocol*

We formalize refactoring as a constrained program transformation problem. Let $x$ denote an input program and let the detector produce a set of smell instances $\mathcal{S}(x) = \{s_1, \ldots, s_k\}$, where each $s_i$ includes (i) a smell type label, (ii) a location span, and (iii) smell-specific metadata. The refactoring generator produces a transformed program $y$ such that edits are restricted to the union of the detector-provided spans, while preserving behavior to the extent feasible without executing tests:

$$y = G(x, \mathcal{S}(x), \pi, \theta), \tag{1}$$

where $G$ is the LLM-guided transformation procedure, $\pi$ denotes the prompt specification (constraints, exemplars, and output schema), and $\theta$ denotes decoding configuration (e.g., temperature). In addition to $y$, the generator returns a structured explanation $e$ that maps each addressed smell instance to the applied transformation rationale. This explicit binding between $\mathcal{S}(x)$ and $(y, e)$ operationalizes controllability: it limits scope drift, enables systematic logging, and supports acceptance-aware configuration selection.

The formulation in Eq. (1) is used directly to define the experimental pipeline and to isolate which components are varied across studies. For each JavaScript artifact $x$, we first compute the deterministic smell report $\mathcal{S}(x)$ via the AST-based detector, and then generate the refactored output (and explanation) $(y, e) = G(x, \mathcal{S}(x), \pi, \theta)$. To quantify refactoring effectiveness, we re-run the same detector on the generated code $y$ to obtain $\mathcal{S}(y)$ and compute smell reduction by comparing $\mathcal{S}(x)$ vs. $\mathcal{S}(y)$ (with post-refactoring non-critical instances tracked separately), alongside maintainability and LoC measurements computed before/after refactoring. Developer acceptance is recorded as a binary decision on each generated pair $(y, e)$, enabling project-level acceptance rates through aggregation.

In the temperature study (Section 3.6), we hold the inputs and control policy fixed, i.e., $(x, \mathcal{S}(x), \pi)$ are unchanged, and vary only the decoding configuration $\theta$ (temperature). This yields outputs $(y_\theta, e_\theta)$ for each candidate temperature, and the acceptance rate is computed as the fraction of artifacts whose generated refactoring is accepted under that $\theta$. The selected operating point $(\theta)$ is then adopted as the default configuration for the full repository evaluation to balance constraint adherence (repeatability) with sufficient flexibility to perform smell-mitigating transformations.

### *3.3 System Architecture*

The prototype is implemented as a microservice-based system, as shown in Fig. 2, comprising five components. The architecture emphasizes modularity, traceability of artifacts, and the ability to evolve individual services (e.g., smell detector, prompt templates, or model backends) independently.

- *Web application:* The client interface supports uploading code, viewing detected smells, previewing refactored code and explanations, and recording accept/reject decisions. It is designed to minimize reviewer burden by presenting the original and refactored versions side-by-side along with the explanation and summary of the smell.
- *Backend orchestrator:* This component acts as the coordination layer and system gateway. Manages request routing, persists refactoring artifacts (original code, smell instances, prompts, model output, explanations) and exposes APIs/webhooks for communication across services. It is also responsible for asynchronous execution when detection and refactoring are triggered in separate steps.

- *Code smell detection service:* This service performs scent parsing and identification using AST-based analysis, based on the defined symptoms described in the Guru resource [22,23]. Returns structured smell instances to the backend to ensure consistent downstream prompt construction and analytics.
- *LLM service:* This service builds controlled prompts, invokes the LLM via an external routing interface, and parses model responses into (i) refactored code and (ii) structured refactoring steps/explanations.
- *Analytics service:* This service retrieves stored refactoring records and computes aggregated metrics, including the percentage of smell reduction, acceptance rate, lines-of-code changes, and the improvement of the maintainability index (MI).

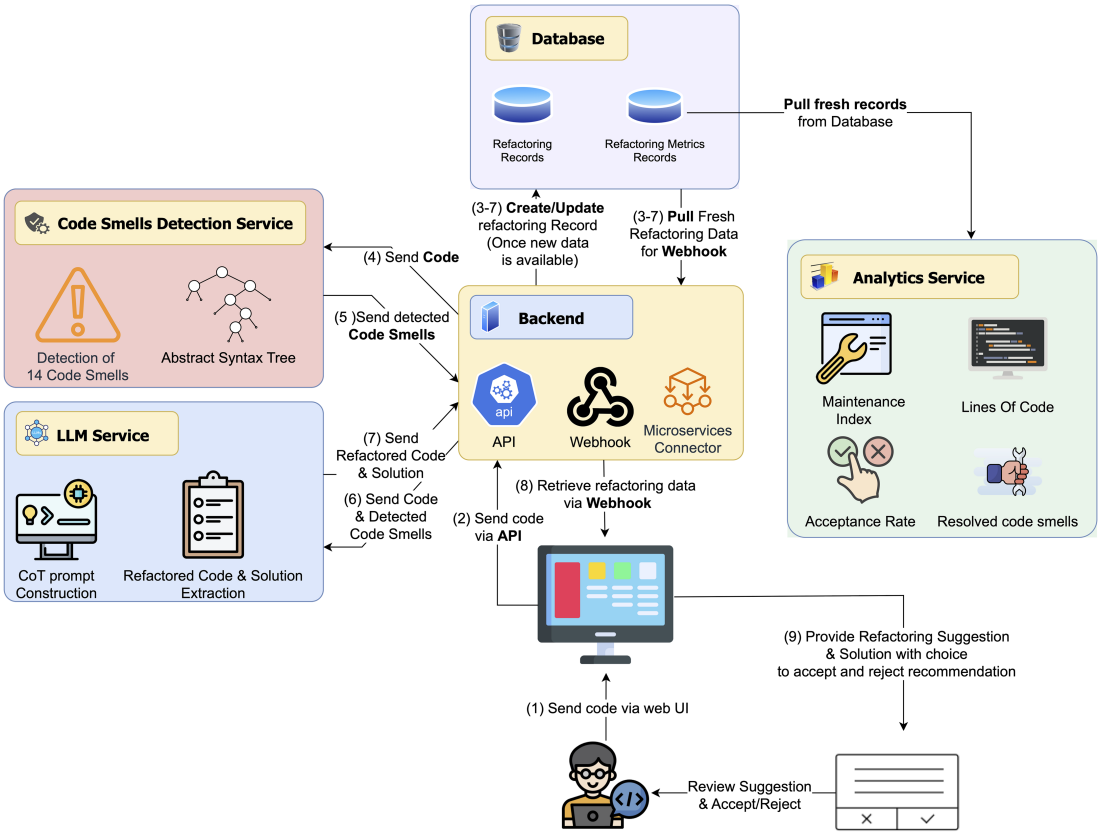


**Figure 2:** System architecture of the proposed refactoring framework.

By decoupling services, the system allows for independent tuning of detection thresholds, prompt templates, and LLM configuration while retaining a stable record schema. This is particularly important for repeatability: changes to model selection or prompt design can be evaluated without re-engineering the smell detector or the analytics pipeline.

### *3.4 Code Smell Detection*

Code smell detection is implemented using deterministic AST-based rules. The input JavaScript code is parsed into an abstract syntax tree, after which smell-specific detectors traverse relevant nodes (e.g., function declarations, class bodies, call expressions) to check conditions that correspond to smell symptoms.

The implementation currently supports 14 code smell categories (Table 1), selected to reflect common maintainability issues observed in JavaScript repositories (e.g., overly long functions, overly large classes,

complex conditional logic, excessive parameter lists, and patterns that degrade readability or modularity). Each smell instance is encoded as a structured record containing: Smell type, Location reference, Supporting metadata required for refactoring, such as counts (e.g., statements/methods), identifier names, and any threshold-triggering attributes that explain why the smell was flagged. Listing 1 shows an illustrative example (fields may vary by smell type).

**Table 1:** Code smell types and operational definitions used by the detector.

| No. | Smell | Operational Definition (Detector Rule) |
|---|---|---|
| 1 | Long Function | Function length >20 lines of code. |
| 2 | Large Class | Class contains >10 methods. |
| 3 | Duplicate Code | Identical or near-identical fragments repeated. |
| 4 | Excessive Parameters | Function/method has >4 parameters. |
| 5 | Magic Numbers | Literals used directly instead of named constants. |
| 6 | Deeply Nested Code | Nesting depth >3 levels. |
| 7 | Unused Variables | Declared variables never referenced. |
| 8 | Primitive Obsession | Repeated literals; boolean flags as args; long string-comparison chains. |
| 9 | Data Clumps | Same parameter group recurs across sites. |
| 10 | Feature Envy | Method accesses another object >3 times. |
| 11 | Middle Man | Method delegates with no local logic. |
| 12 | Switch Statements | Switch has >4 branches/conditions. |
| 13 | Shotgun Surgery | Function modifies properties of more than one object. |
| 14 | Message Chains | Chain depth >3 levels. |

**Listing 1** Example smell-instance record returned by the detector.

```
{
      "smell_type": "LongFunction",
      "location": { "file": "src/utils/parser.js", "start_line": 42, "
        end_line": 118 },
      "target": { "function_name": "parseInput" },
      "metrics": { "loc": 77, "nesting_depth": 2, "param_count": 2 },
      "thresholds": { "loc_gt": 20 },
      "evidence": { "statement_count": 41 },
      "refactoring_hints": ["extract_method", "introduce_helper"]
   }
```

The detector is designed to be extended by (i) implementing a new rule, (ii) registering its output schema so that the backend and LLM service can interpret it consistently, and (iii) adding the corresponding refactoring guidance template used during prompt construction. This ensures that expanding the smell coverage does not require redesigning the pipeline logic.

*Duplicate code smell: scope of detection*

In this work, *Duplicate Code* detection targets **syntactic clones** within a repository (Type-1/Type-2 style duplication), identified by normalizing AST subtrees and matching repeated normalized structures above a minimum size threshold. The detector does not attempt to prove semantic equivalence (Type-3/Type-4

clones) and therefore reports duplication conservatively. This design is consistent with our goal of smell-targeted refactoring under strong scope control, where false positives can increase review burden and false negatives reduce measured smell coverage.

### *3.5 LLM Refactoring Service and Controlled Prompting*

The refactoring step treats LLM generation as a constrained transformation task rather than open-ended rewriting. The goal is to obtain refactorings that are both (a) effective in mitigating detected smells and (b) acceptable for developer review in terms of scope, clarity, and perceived safety.

To enforce this, prompt templates implement four control mechanisms:

- *Scope control:* The model is instructed to refactor only the smell instances reported by the detector. This couples the edit budget to explicit targets and reduces opportunistic changes not related to the smell report.
- *Behavior-preservation intent:* The prompt explicitly discourages the introduction of new features or externally observable changes, framing the task as refactoring rather than redesigning or bug-fixing.
- *Structured output:* The model is required to return (i) refactored code in a parseable block, and (ii) an explanation section that references each smell addressed. This improves automated parsing, logging, and review usability [20,21].
- *Refactoring knowledge grounding:* The prompt includes smell-specific "how-to-fix" exemplars (compact before/after patterns). These examples reduce ambiguity and guide the model towards standard refactoring moves consistent with the target smell [24–27].

After the model returns an output, the LLM service extracts the refactored code and refactoring steps/explanations and sends them back to the backend for presentation in the UI and storage in the database.

*Prompt policy $\pi$: constraints, exemplars, and output schema*

The prompt policy $\pi$ is composed of three elements: (i) scope constraints, (ii) smell-specific exemplars, and (iii) a structured output schema. Listing 2 shows a condensed illustrative excerpt.

**Listing 2** Condensed example of prompt policy $\pi$.

```
You are refactoring JavaScript code. Task: fix ONLY the detected smells
listed below.
   Scope constraints:
   - Do not change external behavior.
   - Do not modify code outside the target span(s) unless required by
the targeted smell.
   - Do not add new features or remove functionality.
   Detected smells:
   1) LongFunction at src/utils/parser.js:42-118 (LOC=77 > 20)
   Allowed refactor patterns (example):
   BEFORE: function parseInput(...) {...long body... }
   AFTER: function parseInput(...) { helper1(...); helper2(...); }
   Output schema (MUST follow):
   [REFACTORED_CODE]
```

```
...
[/REFACTORED_CODE]
[EXPLANATION]
- Smell: LongFunction (42-118): extracted helper functions X/Y;
reduced nesting;
preserved control flow.
[/EXPLANATION]
```

### 3.6 *Temperature Study (Configuration Selection)*

The LLM decoding temperature affects both output variability and the risk of producing changes beyond the intended refactoring scope. To select a practical operating point, we performed a targeted temperature study in which multiple temperature values were evaluated under a fixed prompt structure. The outputs were reviewed for acceptability and alignment with the refactoring intention.

The configuration that produced the highest observed acceptance in this study, as shown in Fig. 3, was approximately 0.4, achieving ≈95% acceptance. This setting was therefore adopted as the default for the broader repository evaluation to balance determinism (repeatability and constraint adherence) with sufficient flexibility for stylistic improvements (e.g., improved naming and decomposition).

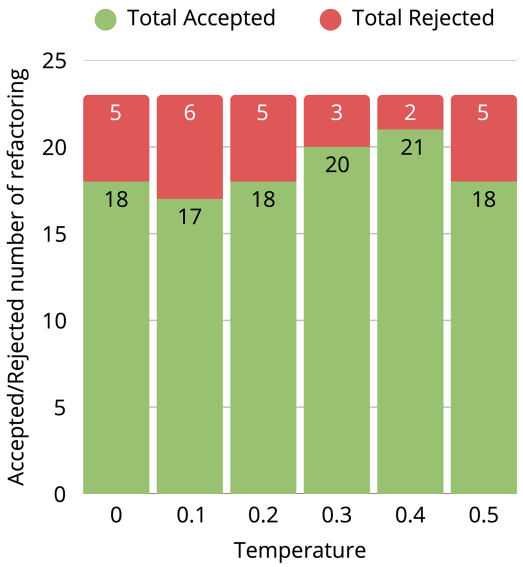


**Figure 3:** Acceptance rate by decoding temperature.

*Acceptance Decision Protocol:*

Because behavior-preserving refactoring must remain reviewable, each generated refactoring suggestion is evaluated through an explicit accept/reject decision during review. An *accepted* suggestion is one judged suitable for integration under the repository's expected coding conventions and perceived behavioral safety, while a *rejected* suggestion is one where the reviewer identified concerns such as potential semantic drift, convention mismatch, or incomplete mitigation of the targeted smell instance(s). These decisions are recorded alongside the original code, the detected smell report, the refactored output and the explanation text, allowing acceptance rate computation and subsequent qualitative analysis of the error modes.

### 3.7 Evaluation Corpus and Protocol

The evaluation uses JavaScript code from two complementary sources: (i) LeetCode-style solutions, which provide controlled, smaller-scale inputs with minimal environmental dependencies, and (ii) multiple GitHub repositories, which reflect realistic variation in coding style, project organization, and smell prevalence. The five repositories used in this study are publicly available: online-exam-portal-dev [28], js-projects-github-1 [29], javascript-code-snippets [30], leetcode-problems [31], ShopingKaro-master [32].

For each repository, the pipeline is applied to *eligible* JavaScript artifacts, defined as inputs that (a) can be parseable by the AST pipeline and (b) contain at least one detected smell instance according to Table 1. For each eligible artifact, the system generates a refactoring suggestion, logs the detector output and model response, and records an accept/reject decision in the refactoring record store. The Repository-level counts are then computed from these stored records, ensuring traceability from aggregate metrics back to individual refactoring events.

### 3.8 Outcome Measures and Computation

We report three outcomes families computed from persisted refactoring records: smell reduction, maintainability change, and practical usefulness (developer acceptance and user feedback). To ensure that the reported values are reproducible and interpretable across heterogeneous repositories, we define the computation and aggregation procedures below.

*Smell counts:* For each analyzed file (or snippet), the detector returns a set of smell instances. The Project-level smell counts are obtained by summing instances of all analyzed artifacts in that repository.

*Noncritical smells:* In some contexts, a detector-triggering pattern may be acceptable or idiomatic (e.g., fluent initialization sequences that resemble message chains). Such instances are marked as *noncritical* after refactoring and are reported separately to avoid overstating the residual technical debt.

*Smell reduction (%):* For each repository, the percentage of decrease is computed as:

$$\text{Decrease} = \frac{S_{\text{orig}} - (S_{\text{after}} - S_{\text{noncrit}})}{S_{\text{orig}}} \times 100,$$

where $S_{\text{orig}}$ is the total number of detected smells in the original code, $S_{\text{after}}$ is the total number of detected smells after refactoring, and $S_{\text{noncrit}}$ is the subset of smells post-refactoring labeled non-critical. The overall average reduction in smell (83.96%) is calculated as the arithmetic mean of the decreases at the repository-level in the evaluated projects.

*Maintainability (MI) and LoC:* Maintainability is assessed using the maintainability index (MI) in conjunction with descriptive LoC statistics. MI and LoC are computed using the same analysis procedure before and after refactoring for each analyzed artifact; repository-level MI is aggregated over artifacts to produce the reported project MI values. The mean MI improvement reported (+5.366) is the arithmetic mean of the MI differences at the repository-level between the evaluated projects. We report MI as a static proxy for maintainability. MI is computed using the standard form:

$$MI = 171 - 5.2\ln(V) - 0.23CC - 16.2\ln(LOC),$$

where $V$ is Halstead volume, $CC$ is cyclomatic complexity, and $LOC$ is lines of code. Values are then normalized to the reporting scale used by our analysis tool. We compute MI consistently before and after refactoring using the same implementation and configuration across all projects.

*Developer acceptance rate:* Practical usefulness is measured as the acceptance rate of refactoring suggestions:

$$\text{Acceptance rate} = \frac{N_{\text{accepted}}}{N_{\text{total}}} \times 100,$$

and, the total mean acceptance rate is calculated as the arithmetic mean of the acceptance rates of the repository.

## 4 Results and Analysis

### *4.1 Code Smell Reduction across Repositories*

The evaluation aims at a fixed set of smells relevant to maintainability with explicit operational thresholds, allowing consistent pre/post comparisons across repositories. The detector covers both size/complexity smells (e.g., Long Function >20 LOC, Deeply Nested Code >3 levels) and structural/design smells (e.g., Feature Envy, Message Chains), which is important because smell prevalence in JavaScript projects is typically heterogeneous: smaller repositories often concentrate in size- and duplication-related smells, whereas larger repositories may also exhibit architecture-related idioms such as chaining and delegation.

Smell-type counts are derived directly from the detector's structured smell-instance outputs logged during evaluation. Because counts depend on operational thresholds (Table 1) and repository snapshots, we compute Table 2 from the stored smell records rather than estimating them from secondary sources.

**Table 2:** Smell types detected per project. A checkmark indicates that at least one instance of the smell type was detected in the project under the operational thresholds in Table 1.

| Smell Type | Exam | JS-Proj | Snip | Leet | Shop |
|---|---|---|---|---|---|
| Long Function | ✓ | ✓ | ✓ | ✓ | ✓ |
| Large Class | ✓ | ✓ | ✓ | – | – |
| Duplicate Code | ✓ | ✓ | ✓ | – | – |
| Excessive Parameters | ✓ | ✓ | ✓ | ✓ | – |
| Magic Numbers | ✓ | ✓ | ✓ | ✓ | ✓ |
| Deeply Nested Code | ✓ | ✓ | ✓ | ✓ | – |
| Unused Variables | ✓ | ✓ | ✓ | ✓ | ✓ |
| Primitive Obsession | ✓ | ✓ | ✓ | ✓ | – |
| Data Clumps | ✓ | ✓ | ✓ | – | – |
| Feature Envy | ✓ | ✓ | ✓ | – | – |
| Middle Man | ✓ | ✓ | ✓ | – | – |
| Switch Statements | ✓ | ✓ | ✓ | ✓ | – |
| Shotgun Surgery | ✓ | ✓ | ✓ | – | – |
| Message Chains | ✓ | ✓ | ✓ | – | – |

Across all evaluated repositories, the framework reduced detected smells. Aggregated across repositories, it achieved an 83.96% average code-smell reduction. The project-level outcomes are summarized in Table 3 where *Original* denotes the total number of detected smell instances aggregated in the 14 smell categories in Table 1. *After refactoring* denotes the same total computed on the refactored output, and *Non-critical* denotes the subset of post-refactoring smell instances labeled non-critical. This pattern suggests that the constrained refactoring process was able to produce changes that directly mitigate detector-triggering

symptoms across multiple smell categories, rather than optimizing for a narrow subset (e.g., formatting or superficial renaming).

**Table 3:** Repository-level smell analysis (pre/post refactoring).

| Project | Original | After | Non-Critical | Decrease |
|---|---|---|---|---|
| online-exam-portal-dev | 262 | 64 | 43 | 92.0% |
| js-projects-github-1 | 225 | 81 | 53 | 87.5% |
| javascript-code-snippets | 172 | 61 | 35 | 84.8% |
| leetcode-problems | 153 | 61 | 48 | 91.5% |
| ShopingKaro-master | 25 | 23 | 14 | 64.0% |

Repository-level variation is expected for three reasons. First, repositories differ in baseline smell distribution: projects with many short files tend to contain fewer long-function and large-class smells, whereas multi-responsibility modules more frequently trigger size and nesting thresholds. Second, certain smells (e.g., Message Chains, Middle Man) can be partially idiomatic under project-specific styles or frameworks, and strict elimination may conflict with local conventions even when the detector flags an instance. Third, the framework is intentionally conservative with respect to behavior preservation and, therefore, prioritizes smell-targeted edits over broad restructuring; consequently, smells that are tightly coupled to domain workflows may be reduced rather than aggressively eliminated depending on context. Overall, the consistent direction of change between projects supports the effectiveness of coupling explicit detection with constrained LLM-based refactoring.

### *4.2 Maintainability Impact*

To assess whether reductions in smell counts correspond to larger improvements in structural quality, maintainability was evaluated using MI, along with descriptive LoC statistics (Table 4). In all repositories, MI improved by +5.366 points on average, indicating that the changes were associated with higher maintainability as captured by this standard proxy.

**Table 4:** Maintainability index (MI) and lines of code (LoC) before/after refactoring.

| Project | Original (LoC total/LoC avg/MI) | After (LoC total/LoC avg/MI) |
|---|---|---|
| online-exam-portal-dev | 8821/73/72.40 | 7114/62/75.41 |
| js-projects-github-1 | 1822/29/73.69 | 2260/37/76.65 |
| javascript-code-snippets | 1102/47/67.21 | 1346/58/71.57 |
| leetcode-problems | 619/28/56.42 | 619/28/66.99 |
| ShopingKaro-master | 944/72/72.28 | 552/42/78.21 |

Table 4 also illustrates why LoC should not be interpreted as a quality proxy in isolation. In some repositories, LoC decreased after refactoring, consistent with elimination of redundancy or simplification of control flow. For example, *online-exam-portal-dev* reduced the total LoC from 8821 to 7114 while the MI increased from 72.40 to 75.41. Similarly, *ShopingKaro-master* reduced the total LoC from 944 to 552, with MI improving from 72.28 to 78.21. These outcomes are consistent with refactorings that consolidate duplication, simplify conditional logic, or decompose overly long functions.

In contrast, other repositories increased in LoC while still improving maintainability. *js-projects-github-1* increased from 1822 to 2260 total LoC (avg per file 29 to 37) while MI improved from 73.69 to 76.65. *javascript-code-snippets* increased from 1102 to 1346 total LoC (avg per file 47 to 58) with MI improving from 67.21 to 71.57. Such patterns are consistent with refactorings that make structure explicit (e.g., extracting helper functions, introducing intermediate variables to reduce expression complexity, replacing "magic numbers" with named constants, or grouping data clumps into objects), which can increase line count while reducing cognitive burden and improving maintainability.

A notable case is *leetcode-problems*, where the total LoC and average LoC per file remain unchanged (619/28 both before and after), yet the MI increases from 56.42 to 66.99. This suggests that the modifications primarily improved structural properties captured by MI without materially changing the code size (e.g., reducing nesting, clarifying expressions, or improving decomposition within existing bounds). Taken together, the MI outcomes indicate that smell-targeted refactoring corresponds to measurable maintainability gains, while LoC changes remain context-dependent and descriptive rather than determinative.

### *4.3 Acceptance Rate and Operating Point*

While static metrics reflect structural change, developer acceptance provides a practical indicator of suitability: refactorings must be understandable, consistent with repository conventions, and perceived as safe with respect to behavior preservation. In all repositories, the mean acceptance rate was 91.66%, indicating that the majority of suggestions were considered reviewable and appropriate.

The Project-level results are shown in Table 5. *online-exam-portal-dev* achieved 118/125 accepted (94.4%); *js-projects-github-1* achieved 58/61 (95%); *leetcode-problems* achieved 20/22 (90%); and *ShopingKaro-master* achieved 12/13 (92%). The lowest acceptance rate among the evaluated repositories was *javascript-code-snippets* at 20/23 (86.9%), which remains high, but suggests greater sensitivity to local conventions or a higher proportion of borderline cases.

**Table 5:** Refactoring suggestion acceptance by project.

| Project Title | Total | Accepted | Rejected | Acceptance Rate |
|---|---|---|---|---|
| online-exam-portal-dev | 125 | 118 | 7 | 94.4% |
| js-projects-github-1 | 61 | 58 | 3 | 95% |
| javascript-code-snippets | 23 | 20 | 3 | 86.9% |
| leetcode-problems | 22 | 20 | 2 | 90% |
| ShopingKaro-master | 13 | 12 | 1 | 92% |

The temperature study provides a complementary explanation for these results. Acceptance peaked around temperature ≈0.4, reaching approximately 95% in controlled trials (Fig. 3). This operating point reflects a balance between determinism and flexibility: at lower temperatures, outputs can be overly conservative and insufficiently responsive to the full smell context (e.g., minimal edits that do not adequately restructure a long function or nested logic), whereas higher temperatures increase the likelihood of stylistic drift or ancillary edits beyond the smell scope, which can trigger reviewer hesitation even when functionality may remain intact. Selecting 0.4 therefore supports stable acceptance while maintaining enough adaptability to generate context-appropriate refactorings across repositories.

### *4.4 Error Modes and Qualitative Patterns*

Although acceptance is high in general, the rejected cases (Table 5) are analytically informative because they illustrate the practical boundaries of LLM-based refactoring when behavior preservation and project conventions matter. The observed rejection drivers fall into three recurring categories. The error modes are derived from manual inspection of rejected and borderline refactoring cases, cross-referenced with detector outputs, reviewer notes recorded during the accept/reject protocol, and automated check failures where applicable.

*(1) Potential Behavioral Drift:*

Some transformations were structurally plausible and improved readability, but raised concerns about subtle JavaScript semantics (e.g., evaluation order, mutation side effects, type coercion, or short-circuit behavior). In such cases, reviewers tended to reject changes unless they were accompanied by stronger validation signals (e.g., tests). This reflects conservative review norms in which refactoring is expected to preserve behavior and uncertainty often triggers rejection.

*(2) Incomplete Smell Resolution:*

A subset of rejected or partially accepted suggestions reduced the intensity of the smell but did not completely eliminate the smell instance according to the operational definition of the detector (Table 1). This may occur when symptoms are entangled with local design constraints (e.g., complex domain workflows within a single function) or when complete elimination would require broader redesign that the framework intentionally avoids. Thus, conservatism in scope control can yield partial mitigation in edge cases.

*(3) Project-Specific Conventions:*

Some rejections reflect mismatch with repository style or architectural preferences rather than refactoring errors. Projects may prefer fluent chaining, specific naming conventions, or minimal function extraction. Even when a change aligns with general refactoring guidance, it may be rejected if it diverges from local norms. This interpretation is consistent with the comparatively lower acceptance rate in *javascript-code-snippets* (86.9%, Table 5), where curated examples can make stylistic deviations more salient.

These patterns suggest clear, incremental avenues to improve adoption without changing the quantitative outcomes reported. Integrating validation hooks (tests, linting, and type checks) would reduce the uncertainty around behavioral preservation and could convert some risk-based rejections into acceptances. In addition, repository-conditioned prompting (e.g., incorporating local formatting rules or style exemplars) could reduce convention mismatch and improve perceived fit, while preserving the detector–LLM workflow underlying the reported improvements.

### *4.5 User Feedback and Adoption Signal*

Quantitative improvements do not guaranty routine adoption. Developers may remain hesitant to integrate automated refactoring into active projects due to workflow disruption, perceived risk, or governance constraints. Table 6 therefore provides a complementary view of perceived output quality. Feedback was collected via a short questionnaire administered after participants inspected refactoring suggestions in the prototype UI. Participants were asked to rate perceived (i) readability improvement, (ii) code simplicity, (iii) maintainability, and (iv) willingness to integrate the system into their projects. Responses were recorded as binary positive/negative judgments per criterion and aggregated across respondents to compute the percentages. In total, $N = 20$ respondents participated, all with prior JavaScript development experience. The respondents reported strong positive feedback for readability (92%), code simplicity (88%), and maintainability (95%). However, willingness to integrate the system into a project was lower (40%). This gap suggests that while refactoring outputs are viewed favorably in isolation, adoption barriers are more

strongly driven by operational concerns (e.g., CI integration, code review policies, fear of regressions, or preference for manual control). Consequently, future work is motivated less by improving the perceived quality of refactorings and more by improving deployment confidence through automated validation, IDE integration, and project-specific customization.

**Table 6:** User feedback on refactored code quality.

| Criterion | Positive Feedback (%) |
|---|---|
| Readability | 92% |
| Code simplicity | 88% |
| Maintainability | 95% |
| Willingness to integrate the system in a project | 40% |

## 5 Discussion

The empirical results in Section 4 indicate that coupling explicit smell detection with constrained LLM-based generation can yield refactorings that are both measurable and reviewable in practice. Beyond the observed improvements in smell prevalence and maintainability proxies, the primary implication is methodological: reliability in LLM-assisted refactoring depends as much on *control mechanisms* as on model capability. By binding edits to detector-identified targets and embedding smell-specific exemplars, the framework reduces opportunities for scope drift, while the structured explanation output improves auditability during review.

Despite strong acceptance, automated refactoring is likely to remain human-in-the-loop in many teams, particularly when semantic preservation cannot be verified automatically. In practice, adoption depends not only on output quality but also on predictability, traceability, and seamless integration into existing toolchains. While the proposed architecture supports these requirements through persisted records of smells, refactorings, explanations, and decisions, deployment in production workflows would benefit from (i) automated validation hooks (tests, type checks, linters) to reduce uncertainty about behavioral preservation; (ii) stricter diff policies where feasible (e.g., function-level edit boundaries or change budgets); and (iii) repository-configurable style and aggressiveness policies to better align outputs with local conventions.

Several limitations should be considered when interpreting the results. First, the smell detector currently supports 14 smell categories and its thresholds may not match organizational standards in all contexts. Second, the evaluation relies on selected GitHub repositories and LeetCode-style solutions, which may not capture the full complexity of industrial systems (e.g., monorepos, heterogeneous build pipelines, extensive test suites, or legacy constraints). Third, acceptance rate is a practically meaningful metric, but remains inherently context- and reviewer-dependent; acceptance may vary across teams and code review cultures. Finally, LLM behavior can change due to model updates or provider-side modifications, implying that operational stability requires versioning, prompt pinning, and periodic re-validation. Future directions and longer-term scope extensions are summarized in Section 7.

## 6 Threats to Validity

Threats to validity further contextualize these findings. *Internal validity* is influenced by the subjectivity of acceptance decisions; multi-reviewer evaluation and inter-rater reliability reporting would strengthen future studies. *Construct validity* is limited by the fact that smell counts depend on detector definitions and may not fully capture design quality; similarly, MI is a proxy that does not represent all dimensions

of maintainability in practice. *External validity* is constrained by the limited set of evaluated projects and the single-language focus; generalization to other project types, ecosystems, and programming languages remains untested. *Reliability* is affected by stochastic decoding; although the temperature study identifies a stable region around ≈0.4, reproducibility in practice requires fixed model versions, pinned prompts, and consistent preprocessing.

## 7 Conclusion and Future Scope

This paper presented a smell-targeted, scope-bounded refactoring framework for JavaScript that couples deterministic AST-based smell detection with constrained LLM-based transformation and a structured output contract (refactored code plus per-smell rationale). By separating detection (*what to change*) from generation (*how to change*) and binding edits to detector-reported smell instances, the framework improves controllability and reviewability of LLM-assisted refactoring. Implemented as a microservice-based prototype, the approach was evaluated on LeetCode-style solutions and multiple GitHub repositories, achieving an average smell reduction of 83.96%, an average maintainability index improvement of +5.366, and a mean developer acceptance rate of 91.66%. A targeted temperature study further identified an acceptance-optimized operating point around 0.4 (approximately 95% in controlled trials), supporting practical configuration guidance for reliable, human-in-the-loop refactoring.

An important extension is to move from smell-local refactoring toward refactoring guided by domain knowledge expressed in domain-specific models (DSMs) and domain-specific (including graphical) modeling languages. In such settings, refactoring constraints and acceptable transformations can be derived from higher-level specifications (e.g., architectural rules, domain invariants, and model consistency constraints), enabling controlled regeneration of source code across languages, including JavaScript. A complementary direction is to use LLM feedback to support model and language refinement, for example by identifying recurring design deviations in code and translating them into candidate modeling constraints or model refactorings. While this work focuses on smell-targeted program transformations at the code level, the same scope-control and traceability principles can serve as an interface layer between model-level specifications and LLM-driven code transformations. Future work is therefore motivated along two complementary tracks. In the near term, expanding smell coverage and enabling configurable smell policies would broaden applicability, while IDE-level integration could reduce workflow friction. In parallel, incorporating automated behavioral validation via tests and static checks would address the main driver of risk-based rejection. Longer-term directions include multilingual refactoring support, repository-conditioned prompting to better match local styles, and adaptive policies that tune refactoring aggressiveness based on code context and developer feedback histories.

**Acknowledgement:** The authors thank the colleagues who provided feedback on the evaluation protocol and manuscript clarity.

**Funding Statement:** The authors received no specific funding for this study.

**Author Contributions:** Conceptualization, Emir Kuanyshev and Hashim Ali; methodology, Emir Kuanyshev and Hashim Ali; software, Emir Kuanyshev; validation, Emir Kuanyshev and Hashim Ali; formal analysis, Emir Kuanyshev; investigation, Emir Kuanyshev; writing—original draft preparation, Emir Kuanyshev; writing—review and editing, Emir Kuanyshev and Hashim Ali; supervision, Hashim Ali. All authors reviewed and approved the final version of the manuscript.

**Availability of Data and Materials:** Not applicable.

**Ethics Approval:** Not applicable.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References


1. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. Refactoring: improving the design of existing code. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 1999.
2. AlOmar EA, AlRubaye H, Mkaouer MW, Ouni A, Kessentini M. Refactoring practices in the context of modern code review: an industrial case study at xerox. In: Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '21. Piscataway, NJ, USA: IEEE; 2021. p. 348–57. doi:10.1109/ICSE-SEIP52600.2021.00044.
3. Hou X, Zhao Y, Liu Y, Yang Z, Wang K, Li L, et al. Large language models for software engineering: a systematic literature review. ACM Trans Softw Eng Methodol. 2024;33(8):1–79. doi:10.1145/3695988.
4. Bui TD, Vu TT, Nguyen TT, Nguyen S, Vo HD. Correctness assessment of code generated by large language models using internal representations. J Syst Softw. 2025;230(3):112570. doi:10.1016/j.jss.2025.112570.
5. Pomian D, Bellur A, Dilhara M, Kurbatova Z, Bogomolov E, Sokolov A, et al. EM-Assist: safe automated extract method refactoring with LLMs. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024. New York, NY, USA: Association for Computing Machinery; 2024. p. 582–6. doi:10.1145/3663529.3663803.
6. Ferreira F, Valente MT. Detecting code smells in React-based Web apps. Inf Softw Tech. 2023;155(4):107111. doi:10.1016/j.infsof.2022.107111.
7. Gao S, Wen XC, Gao C, Wang W, Zhang H, Lyu MR. What makes good in-context demonstrations for code intelligence tasks with LLMs?. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway, NJ, USA: IEEE; 2023. p. 761–73.
8. Feng S, Chen C. Prompting is all you need: automated android bug replay with large language models. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24. New York, NY, USA: Association for Computing Machinery; 2024. p. 803–15. doi:10.1145/3597503.3608137.
9. Cui X, Wu JZ, Luo TY, Ling X, Wang X. An LLM-based framework for README generation via code-aware representation and dual-stage optimization. J Comput Res Dev. 2026;63(4):918–42. (In Chinese). doi:10.7544/issn1000-1239.202550698.
10. Jiang J, Wang F, Shen J, Kim S, Kim S. A survey on large language models for code generation. ACM Trans Softw Eng Methodol. 2026;35(2):1–72. doi:10.1145/3747588.
11. Gao Y, Hu X, Yang X, Xia X. Automated unit test refactoring. arXiv:2409.16739. 2025.
12. Taentzer G, Arendt T, Ermel C, Heckel R. Towards refactoring of rule-based, in-place model transformation systems. In: Proceedings of the First Workshop on the Analysis of Model Transformations, AMT '12. New York, NY, USA: Association for Computing Machinery; 2012. p. 41–6. doi:10.1145/2432497.2432506.
13. Markovič L. Towards rule based refactoring. In: IIT.SRC 2016: 12th Student Research Conference in Informatics and Information Technologies; 2016 Apr 28; Bratislava, Slovakia.
14. Zhang J, Nie P, Li JJ, Gligoric M. Multilingual code co-evolution using large language models. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery; 2023. p. 695–707. doi:10.1145/3611643.3616350.
15. Santos R, Santos I, Magalhaes C, de Souza Santos R. Are we testing or being tested? Exploring the practical applications of large language models in software testing. In: 2024 IEEE Conference on Software Testing, Verification and Validation (ICST). Piscataway, NJ, USA: IEEE; 2024. p. 353–60.
16. Wang J, Chen Y. A review on code generation with LLMs: application and evaluation. In: 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI). Piscataway, NJ, USA: IEEE; 2023. p. 284–9.
17. Cummins C, Seeker V, Armengol-Estapé J, Markosyan AH, Synnaeve G, Leather H. Don't transform the code, code the transforms: towards precise code rewriting using LLMs. arXiv:2410.08806. 2024.
18. Mirchev M, Costea A, Singh AK, Roychoudhury A. Assured automatic programming via large language models. arXiv:2410.18494. 2024.

19. Zhang Z, Wang Y, Wang C, Chen J, Zheng Z. LLM hallucinations in practical code generation: phenomena, mechanism, and mitigation. arXiv:2409.20550. 2024.
20. Dehghan S. Assessing code reasoning in large language models: a literature review of benchmarks and future directions. Preprints. 2024 doi:10.20944/preprints202411.1147.v1.
21. Weyssow M, Zhou X, Kim K, Lo D, Sahraoui H. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. ACM Trans Softw Eng Methodol. 2025;34(7):204. doi:10.1145/3714461.
22. Sharma T. Detecting and managing code smells: research and practice. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE '18. New York, NY, USA: Association for Computing Machinery; 2018. p. 546–7. doi:10.1145/3183440.3183460.
23. Refactoring Guru. Code smells. 2019 [cited 2025 Mar 2]. Available from: https://refactoring.guru/refactoring/smells.
24. Schulhoff S, Ilie M, Balepur N, Kahadze K, Liu A, Si C, et al. The prompt report: a systematic survey of prompt engineering techniques. arXiv:2406.06608. 2025.
25. Coello CEA, Alimam MN, Kouatly R. Effectiveness of ChatGPT in coding: a comparative analysis of popular large language models. Digital. 2024;4(1):114–25.
26. Coignion T, Quinton C, Rouvoy R. A performance study of LLM-generated code on leetcode. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE '24. New York, NY, USA: Association for Computing Machinery; 2024. p. 79–89. doi:10.1145/3661167.3661221.
27. Zhao J, Yang D, Zhang L, Lian X, Yang Z, Liu F. Enhancing automated program repair with solution design. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24. New York, NY, USA: Association for Computing Machinery; 2024. p. 1706–18. doi:10.1145/3691620.3695537.
28. Abdullah M. Online-Exam-Portal-Dev; 2026 [cited 2025 Mar 2]. Available from: https://github.com/abdullahthewebbee/Online-exam-portal-dev.
29. Pathak P. Js-Projects-Github-1; 2026 [cited 2025 Mar 2]. Available from: https://github.com/zpratikpathak/25-Javascript-Projects-for-beginner.
30. roeib. Javascript-Code-Snippets; 2026 [cited 2025 Mar 2]. Available from: https://github.com/roeib/JavaScript-snippets.
31. Crozier J. Leetcode-Problems; 2026 [cited 2025 Mar 2]. Available from: https://github.com/topics/leetcode-problems.
32. Arya A. ShopingKaro-master; 2025 [cited 2025 Mar 2]. Available from: https://github.com/itzabhinavarya/ShopingKaro.