BitsAI-CR: Automated Code Review via LLM in Practice

Tao Sun, Jian Xu[†], Yuanpeng Li, Zhao Yan, Ge Zhang, Lintao Xie, Lu Geng, Zheng Wang,
Yueyan Chen, Qin Lin, Wenbo Duan, Kaixin Sui

ByteDance

Abstract

Code review remains a critical yet resource-intensive process in software development, particularly challenging in large-scale industrial environments. While Large Language Models (LLMs) show promise for automating code review, existing solutions face significant limitations in precision and practicality. This paper presents BitsAI-CR, an innovative framework that enhances code review through a two-stage approach combining RuleChecker for initial issue detection and ReviewFilter for precision verification. The system is built upon a comprehensive taxonomy of review rules and implements a data flywheel mechanism that enables continuous performance improvement through structured feedback and evaluation metrics. Our approach introduces an Outdated Rate metric that can reflect developers' actual adoption of review comments, enabling automated evaluation and systematic optimization at scale. Empirical evaluation demonstrates BitsAI-CR's effectiveness, achieving 75.0% precision in review comment generation. For the Go language which has predominant usage at ByteDance, we maintain an Outdated Rate of 26.7%. The system has been successfully deployed at ByteDance, serving over 12,000 Weekly Active Users (WAU). Our work provides valuable insights into the practical application of automated code review and offers a blueprint for organizations seeking to implement automated code reviews at scale.

Keywords

Code Review, Large Language Model, Data Flywheel

1 Introduction

Code review is a critical process in software development that significantly impacts code quality and project success [2, 48]. It identifies potential security vulnerabilities, logical errors, and performance bottlenecks while facilitating knowledge sharing among R&D teams[34]. However, our analysis at ByteDance reveals significant challenges in enterprise-scale code review practice. The internal data shows that reviewers spend an average of 15 minutes per review for over 50% of cases, with the remaining cases taking even longer. Only 30% of reviews receive immediate attention, while over 67% of engineers express a need for more effective tools.

This evidence indicates a pressing demand for effective automated code review tools, yet current approaches [8, 35] remain inadequate or lack their practical impact comprehensive evaluation at an enterprise scale. Traditional static analysis tools, while offering basic code quality checks, introduce significant overhead to development workflows through complex build configurations and compilation. Large language models (LLMs) emerge as a promising solution due to their superior capabilities in code understanding and natural language generation [9, 12, 22]. Recent commercial implementations have begun exploring LLM-based approaches to

address these limitations, however, Google's approach [40] only focuses on issue classification without generating specific review comments, while Tencent [50] primarily addresses code maintainability concerns. Our investigation reveals three fundamental challenges in current LLM-based solutions: *i)* insufficient precision in generating technically accurate comments, *ii)* low practicality of comments that are technically correct but fail to provide substantial value [26, 29], and *iii)* lack of systematic mechanisms for targeted improvement, preventing data-driven evolution in both model precision and suggestion practicality.

To address these limitations, we present BitsAI-CR, a framework comprised of two primary components operating in synergy: a high-precision review comment generation pipeline and a data flywheel mechanism that continuously optimizes based on user feedback. First, to tackle the issue of low precision in review comments, the review generation pipeline is designed as a two-stage process called the RuleChecker and the ReviewFilter: (1) RuleChecker is a fine-tuned LLM trained on a comprehensive taxonomy of 219 review rules, aimed at identifying potential issues, (2) ReviewFilter is another fine-tuned LLM followed by RuleChecker which improves precision by validating the detected issues. Second, to enable targeted continuous improvement, we construct a data flywheel that leverages real-world feedback for large-scale industrial scenarios: (1) leveraging the annotation feedback data to enhance training datasets, thereby improving the BitsAI-CR's performance with targeted enhancements; (2) metring the Outdated Rate to resolve the problem of technically correct but practically superfluous comments by automatically measuring the percentage of code lines modified after being flagged by BitsAI-CR, which provides concrete evidence of whether developers accept and act upon the review; and (3) dynamically adjusting review rules based on both Outdated Rates and precision measurements, removing rules that generate low-value comments with low Outdated Rates and high precision. Through this approach, BitsAI-CR establishes a continuous improvement cycle to enhance code review.

Empirical evaluation shows that our BitsAI-CR takes significant improvements in precision, and developers are increasingly recognizing the generated comments. The two-stage review pipeline achieves a peak precision of 75.0%, significantly reducing superfluous comments. More importantly, our data flywheel mechanism shows consistent improvement over time, with a 26.7% Outdated Rate in Go. Large-scale industrial deployment at ByteDance further validates BitsAI-CR's impact, engaging over 12,000 Weekly Active Users (WAU).

This paper details how we build and deploy BitsAI-CR, used in the ByteDance production environment for several months and serving ten thousand engineers. The main contributions are:

- A comprehensive taxonomy of review rules that streamlines the entire code review process, enabling evaluation
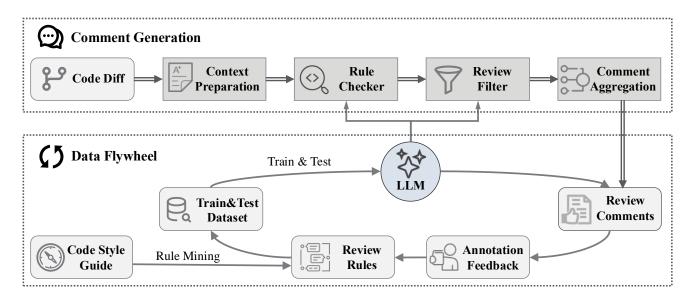

[†] Corresponding Author.

Figure 1: The overview of BITsAI-CR framework for enhancing code review.

practices, data collection mechanisms, and continuous optimization through a data-driven feedback loop.

- A two-stage approach combining RuleChecker for issue detection and ReviewFilter for verification, demonstrating substantial improvements in review effectiveness.
- The introduction of an Outdated Rate metric that addresses two key limitations of traditional precision measurements: the sustainability of manual evaluation and the ability to assess user acceptance of review comments.
- Empirical validation through large-scale deployment, demonstrating the data flywheel's effectiveness and scalability in real-world software development environments through systematic implementation practices.

2 Background

Code review evolves from a simple correctness-checking mechanism to a comprehensive quality assurance process that shapes organizational culture and development practices. Modern reviews now evaluate multiple dimensions of code quality, including design patterns, architectural choices, and maintainability, while fostering knowledge sharing and team collaboration[7]. Meanwhile, code best practices[28] refers to a set of widely accepted methodologies that prove to enhance the quality and efficiency of the software development process. Code review serves as a crucial means to implement these best practices, with reviewers evaluating various aspects of the code based on the company's internal standards.

At ByteDance, code review is an integral part of the software development lifecycle, following industry-standard practices while incorporating company-specific requests. Similar to existing code management systems like GitHub's pull request[17] and Google's Piper[32], ByteDance's software development process requires experienced reviewers to examine every code change submission. The code is only merged after receiving approval for "LGTM" (i.e., "looks

good to me"). Authors and reviewers exchange opinions through the code review system, examining snapshots of affected files.

The typical incremental review process involves several steps: (1) A developer creates a Merge Request (MR). (2) The code diff is submitted to reviewers for error identification and resolution, which can be automated or enhanced by BITsAI-CR. (3) Reviewers, assisted by or replaced with BITsAI-CR, provide feedback to ensure the code meets quality standards. (4) The developer refactors the code based on reviewer feedback. (5) Steps 2-4 are repeated as necessary. (6) Once reviewers confirm all issues are resolved, the code receives approval for merging.

3 Methodology

In this section, we present our approach to automated code review. We first introduce an overview of our framework that illustrates how the Review Comment Generation pipeline and Data Flywheel mechanisms work together as shown in Figure 1. We then detail the taxonomy of code review rules that serve as the foundation for the entire system. Next, we elaborate on the review comment generation pipeline, which includes context preparation, rule checking, review filtering, and comment aggregation. Finally, we describe the data flywheel mechanism to continuously improve BITsAI-CR.

3.1 Framework of BITsAI-CR

The task of BITsAI-CR is to analyze code changes and provide meaningful review comments. Given a code diff, the system aims to identify potential issues and generate appropriate review suggestions. The output includes the review category, specific problematic code locations, and detailed explanatory comments. Figure 2 illustrates this input-output relationship. The input consists of code changes, while the output provides structured review feedback that helps developers improve their code quality.

As shown in Figure 1, the BITsAI-CR framework consists of two primary components: a Review Comment Generation pipeline and

```
diff --git a/calculate.go b/calculate.go
index a123456..b789012 100644
--- a/calculate.go
+++ b/calculate.go
@@ -7,3 +7,5 @@ func CalculateArea(radius float64) float64 {
-    const pi = 3.14
+    const pi = 3.14159
+    area := pi * radious * radious
+    return area
 }
```

Output:

```
Line: 9
Category: Spelling Error
Severity: Medium
Issue: In the 9th line of the modified code, the variable
↪ name 'radious' has a spelling error.
Suggestion: Please correct the spelling error in the variable
↪ name (change 'radious' to 'radius').
```

Figure 2: An Example of Input and Output in BitsAI-CR

a Data Flywheel mechanism for continuous improvement. The Review Comment Generation pipeline processes code reviews through four steps: (1) Context Preparation that structures the input for analysis, (2) RuleChecker that identifies potential issues using an LLM, (3) ReviewFilter that validates the detected issues, and (4) Comment Aggregation that consolidates similar feedback. Supporting this pipeline, the Data Flywheel mechanism ensures continuous improvement through the taxonomy of review rules, which are used to create training and testing datasets for the LLM. The generated review comments receive annotation feedback, which in turn helps refine the review rules, creating a self-improving system that enhances code review quality over time.

3.2 The Taxonomy of Code Review Rules

We establish a comprehensive taxonomy of code review rules which serves as a foundational component of our BitsAI-CR system, and introduce a three-tiered classification structure: review dimensions (broad areas of code quality assessment), review categories (specific issue types requiring evaluation) and review rules (detailed criteria with examples). The term "review rules" refers to detailed specifications that define concrete evaluation criteria. Each review category is backed by detailed rules that specify elements such as issue type classification, severity levels, comprehensive descriptions, and illustrative examples of both proper and improper implementations. Our review dimensions encompass the unique characteristics of various programming languages, including language-specific idioms, performance optimizations, and common pitfalls associated with each language's ecosystem. As shown in Table 1, they include:

- **Code Defect**: This dimension covers various aspects of code correctness, including error handling, logic flaws, and resource management.
- **Security Vulnerability**: It addresses critical security concerns such as injection vulnerabilities, cross-site scripting, and insecure data handling.

Table 1: Distribution and Outdated Rates in the Taxonomy of Code Review Rules

Review Dimension	Review Category	Distribution (%)	Outdated Rate (%)
Security Vulnerability	SQL Injection	0.08	11.54
	Insecure Deserialization	0.03	20.00
	Insecure Object Reference	0.04	15.38
	Memory Leak - Long-term Reference Holding	0.05	37.50
	Improper Password Handling	0.29	28.41
	Type and Non-null Assertion	0.46	14.18
	Cross-Site Scripting (XSS)	0.07	30.43
	Cross-Site Request Forgery (CSRF)	0.10	23.33
Code Defect	Function Parameter Passing	0.71	27.73
	Loop Logic Errors	0.39	33.61
	Database Access Error	0.20	27.87
	Conditional Logic Error/Omission/Duplication	3.48	30.61
	Null Pointer Exception	22.79	27.58
	Algorithm/Business Logic Error	3.50	18.89
	Index/Boundary Condition Error	3.57	26.73
	Syntax Issue	0.08	25.00
	Resource Not Released/Resource Leak	0.60	17.93
	Error and Exception Handling Issue	4.11	25.20
	Incorrect Concurrency Control	0.65	17.91
	Data Format, Conversion, and Comparison Error	1.30	41.37
	Incorrect Sequence Dependency	0.03	0.00
Maintainability and Readability	Unclear Naming	0.34	25.00
	Code Testability Issues	0.05	26.67
	Code Readability	5.00	15.82
	Code Formatting Errors/Inconsistencies	0.60	13.51
	Redundant/Complex Conditional Logic	2.24	20.99
	Variable Naming Conventions	0.61	5.88
	Complex Code	0.80	23.48
	Spelling Error	7.97	31.52
	Unused Definition/Redundant Code	1.83	21.63
	Missing or Inappropriate Code Comments	8.27	26.25
	Overly Long Functions or Methods	2.15	14.76
	Code Duplication	7.23	17.88
	Unclear Error Handling	0.11	17.14
	Magic Numbers/Strings	20.62	18.87
Performance Issue	Inappropriate Data Structures	0.38	13.56
	Unoptimized Loops	0.14	22.73
	Data Format Conversion Performance	0.37	29.57
	Excessive or Improper Lock Usage	0.01	0.00
	Excessive I/O Operations	0.25	13.16
	Repeated Calculations	0.01	0.00

- **Maintainability and Readability**: It emphasizes code clarity, consistency, and structure to ensure the long-term maintainability of the codebase.
- **Performance Issue**: This category focuses on optimizing code execution, covering areas like efficient data structures, query optimization, and resource utilization.

Each Review Category encompasses multiple specific review rules, enabling fine-grained quality assessment. For instance, consider the review dimension "Code Defects", which encompasses various review categories including "Conditional Logic Error/Omission/Duplication". Within this category, specific review rules include "unreachable code detection" and "infinite loop identification", etc. These rules address critical code quality issues: "unreachable code" represents redundant statements that can never be executed due to logical constraints, while "infinite loop identification" identifies potentially endless program execution cycles that could lead to system resource exhaustion. We already include a range of programming languages including Go, JavaScript, TypeScript, Python, and Java, totalling 219 review rules.

This structure offers several key advantages: (1) Efficient Data Collection: The hierarchical taxonomy enables systematic data gathering and labeling, with clear guidelines for categorizing different

types of code issues and their corresponding comments. (2) Streamlined Model Training: The structured categorization provides well-defined training objectives and organized datasets, leading to more effective model fine-tuning for specific review tasks. (3) Systematic Evaluation: The clear classification structure enables precise measurement of model performance across different review dimensions, facilitating comprehensive quality assessment. (4) Targeted Optimization: The structured feedback mechanism allows for systematic improvement of model performance through clearly defined enhancement paths for each review dimension.

3.3 The Pipeline of Review Comment Generation

Our review comment generation pipeline implements a two-stage approach centred on RuleChecker and ReviewFilter. The first stage employs RuleChecker to identify potential issues based on the taxonomy, while the second stage utilizes ReviewFilter to validate these findings, ensuring high precision. Two auxiliary components complete the pipeline: Context Preparation, which structures and enriches the input code for analysis, and Comment Aggregation, which aggregates similar feedback to prevent information overload.

Context Preparation. This phase implements a systematic approach to prepare the input code and construct appropriate prompts for analysis. The process involves three key steps: (1) We partition each code diff based on header hunks, creating several units for analysis to manage context length and prevent excessive token consumption in subsequent processing steps. (2) We expand each segmented code block to include complete function definitions, following a controlled strategy that extends to function boundaries within four times the original diff size, or limits context to three times the original size otherwise. We utilize tree-sitter[1] for precise code block identification and function boundary detection. (3) We implement a detailed change annotation system that marks each line with its status and position: old version lines are marked as "[deleted or pre-modified @line_number in old code]" or [unchanged]", while new version lines are marked as "[added or post-modified @line_number in new code]" or "[unchanged]". These annotations preserve the original structure while providing crucial context about code modifications. When these contexts are prepared, we will apply them to the subsequent prompt construction. This systematic preparation ensures the model receives both comprehensive context and clear instructions, enabling more accurate and relevant review comments.

RuleChecker. The RuleChecker component identifies issues using a fine-tuned LLM that integrates ByteDance's internal code standards with our taxonomy of review rules. This model examines the target code thoroughly and provides detected issues along with modification suggestions, the outputs such as in Figure 2. Additionally, the component includes a rule category blocker that enables dynamic rule exclusion without model retraining when we deem them unacceptable to the user.

ReviewFilter. The ReviewFilter takes a review comment as input and outputs a binary decision (yes/no) to determine whether the

comment should be retained. This component enhances pipeline precision by addressing hallucination and factual errors in the output of RuleChecker. In practice, we reveal that RuleChecker frequently encounters hallucination-related and factual error-related bad cases when identifying issues. However, our experiments find that conventional Supervised Fine-Tuning methods for RuleChecker prove insufficient for error mitigation, even with optimized training samples and reinforcement learning approaches. Through multiple iterations, we eventually opted to introduce the ReviewFilter component to perform a secondary validation on the results generated by RuleChecker, thereby improving the precision of the comments. The ReviewFilter component is also based on a fine-tuned LLM. To enable effective comment validation, we explore three distinct reasoning patterns in the model's output structure: (1) Direct Conclusion, which generates a single decision (Yes or No) token without reasoning; (2) Reasoning-First, which provides complete reasoning before concluding, necessitating full token generation like chains of thought (CoT) [44]; and (3) Conclusion-First, which outputs a decision token followed by supporting rationale, requiring only the first token for evaluation while preserving the complete output. Each pattern offers different trade-offs between precision, inference speed, and explainability, which we evaluate extensively in Section 4.2. We finally chose the Conclusion-First pattern to organize the training data.

Comment Aggregation. Multiple files and header hunks within a single MR may generate lots of similar comments, potentially overwhelming developers with redundant feedback. The comment Aggregation module is at the end of the pipeline. This module employs cosine similarity to determine whether comments are similar and randomly retains one comment from each similar group. Specifically, issue categories and comments are vectorized using our internal embedding model, Doubbao-embedding-large, which operates in 512 dimensions.

After going through the above pipeline, we will obtain a batch of accurate comments for each merge request. These comments will include detailed problem descriptions, modification suggestions, and will be sufficiently aggregated to prevent disruption.

3.4 Evaluation Metrics

Prioritizing Precision in Code Review Practice. In the early stages of code review development, we identify two critical insights through analysis and user studies. First, similar to alert fatigue, developers tend to ignore review comments entirely when faced with numerous comments, as they are unwilling to invest time in careful discrimination. Second, inaccurate comments in the early stages significantly damage user trust, hindering continuous iteration and improvement of the system. These practical challenges lead us to prioritize precision over recall in our approach. Regarding recall, a fundamental challenge lies in the substantial human effort required for comprehensive issue detection. Many defects only manifest after extended periods of production deployment or a large amount of manpower investment, and even well-tested code may harbour latent issues that emerge only through specific usage patterns or edge cases. Formally, let $C_{correct}$ represent the set of correct comments and C_{total} represent all comments generated by BɪᴛsAI-CR,

---

[1]https://tree-sitter.github.io/tree-sitter/

we define precision as:

$$\text{Precision} = \frac{|C_{correct}|}{|C_{total}|} \times 100\% \qquad (1)$$

Outdated Rate for Automated Evaluation. While precision serves as a crucial metric for evaluating model performance, precision measurements alone present fundamental limitations. A primary limitation is that precision cannot reflect whether developers actually accept and act upon the review comments. Additionally, manual precision assessment requires significant effort, making it challenging to conduct both large-scale evaluations and sustained monitoring over time. To address these limitations, particularly the need to measure developer response to comments, we introduce the *"Outdated Rate"* metric, which tracks the percentage of code lines modified after being flagged by BitsAI-CR. Specifically, this metric is computed as:

$$\text{Outdated Rate} = \frac{|\{c \in C_{seen} \wedge \text{isOutdated}(c)\}|}{|C_{seen}|} \times 100\% \quad (2)$$

where C_{seen} represents the set of comments reviewed by code committers within a one-week measurement window, and the function isOutdated(c) returns true only if a comment c is considered outdated, which occurs when if any line within its flagged code range is modified in subsequent commits. This automated measurement approach enables systematic evaluation at a scale where manual assessment becomes impractical. While the Outdated Rate doesn't definitively prove that changes were made in direct response to BitsAI-CR's comments, it helps improve systems continuously in large-scale deployments.

3.5 Data Flywheel: Continuous Evolution

The effectiveness of BitsAI-CR relies on high-quality datasets and continuous feedback-driven iteration. Our data flywheel approach systematically constructs, maintains, and enhances datasets through targeted optimizations based on user feedback as illustrated in the lower part of Figure 1.

3.5.1 Mining Review Rules integrate Code Style Guide. The foundation of BitsAI-CR is a comprehensive taxonomy of code review rules that integrates code style guides with practical review experience (displayed in Table 1 and Section 3.2). Code style guides provide standardized conventions for code formatting, best practices, programming principles, and so on. Based on the code style guide, we derive rules from two primary sources:

- **Internal Static Analysis Rules**: ByteDance employs a comprehensive set of internal rules for static analysis. Each static analysis rule is marked with the recommendation index and acceptance rate. We select those rules with high recommendation index and high developer acceptance rate to form our review rule set.
- **Manual Review Comments**: Considering the deficiencies of Static Analysis Rules in understanding code semantics, we categorize internal manual review comments to extract review rules not covered by static analysis rules. This includes review rules such as "spelling errors", "duplicate code", and "unclear code comments".

Our current system encompasses 219 review rules across five programming languages, dynamically updated through the feedback loop described below.

3.5.2 Dataset Construction for Model Training. The data flywheel transforms review rules into high-quality training datasets through a systematic process:

- **Original Data Collection**: We extracted 120000 code review comments from our internal code repository's MR comments, encompassing both static analysis results and manual review feedback. This original dataset served as the foundation for the primary source for subsequent data refinement.
- **Data Refinement**: We develope a refinement process utilizing the `Doubao-Pro-32K-0828` LLM. For manual review comments, the process first filters non-substantive content (e.g., conversational elements, emojis) and classifies the remaining samples to retain rule-compliant comments. Then enhance these comments by expanding concise feedback into comprehensive issue descriptions with specific modification suggestions. For static analysis results, we refined the comments based on the specific definitions of static analysis rules while incorporating targeted modification suggestions.
- **Quality Assurance**: We ensure dataset quality through systematic manual sampling and annotating. To save labour costs, we employ deterministic sampling rules. After annotating samples from each review rule, we adjust the sampling rates—reducing the rate for high-precision review rules and increasing the rate for low-precision ones.

This process generates refined training datasets comprising approximately 18,000 samples each for Go and front-end languages, and 5,000 samples each for other programming languages.

3.5.3 Online Feedback and Continuous Evolution. After the deployment of BitsAI-CR, we conduct detailed evaluations of its online performance every week and make targeted adjustments based on the evaluation results. Our assessments primarily come from the following three aspects:

- **User Feedback Collection**: The most direct feedback comes from the user's likes and dislikes, often accompanied by reasons for the ratings. By analyzing the like-dislike data, we can specifically optimize the underperforming review rules. However, this type of feedback data is relatively sparse and is generally used as supplementary analysis.
- **Manual Precision Annotation**: Each day, we sample and annotate the online data (sampling generally does not exceed 10%). The annotation results are compiled weekly, providing an accurate measurement of the review rules' precision. For review rules with low precision, we will collect the corresponding samples to retrain the LLM. The training data for the aforementioned RuleFiler component mainly comes from this source.
- **Outdated Rate Monitoring**: Every week, we track changes in the Outdated Rate of each review rule. For scenarios

Figure 3: A Correct but Superfluous Comment

Figure 4: BɪᴛsAI-CR settings interface for enabling review participation and default reviewer invitation.

where the precision is high but the Outdated Rate is consistently low, we consider whether the review rule is not being accepted by users and may decide on decommissioning it.

These diverse feedback channels provide valuable insights into both the technical precision and practical utility of our review rules. For example, Figure 3 demonstrates this with an example where BɪᴛsAI-CR flags the use of a magic number "100". While this suggestion aligns with Go language coding standards that discourage magic numbers, the user responds with a dislike. Such cases that users dislike highlight the need for a more comprehensive evaluation framework beyond simple precision metrics.

To address this challenge, we develop a systematic approach using both precision and Outdated Rates as key metrics for evaluating review rules. Rules are assessed based on specific criteria: an Outdated Rate of around 25% (±5%) with a precision of around 65% (±5%) for 14 days. This dual-metric evaluation framework enables us to make data-driven decisions about retaining or removing specific review rules. Our comprehensive taxonomy of code review rules as shown in Table 1 serves as the foundation for this evolution process. By mapping all online feedback samples to specific rules within this taxonomy, we can efficiently collect data and conduct targeted evaluations. This systematic approach creates a continuous feedback loop: user interactions provide data for rule evaluation, evaluation results guide rule refinements, and refined rules generate new feedback. Through this iterative process, we maintain a dynamic review system that consistently adapts to user needs while maintaining high-quality standards.

3.6 Implementation of BɪᴛsAI-CR

Developers can easily enable BɪᴛsAI-CR features as shown in Figure 4. The **input** to our system is a code diff, and the **output** is a comprehensive review comment as shown in Figure 2. As illustrated in Figure 5, BɪᴛsAI-CR automatically identifies potential issues in
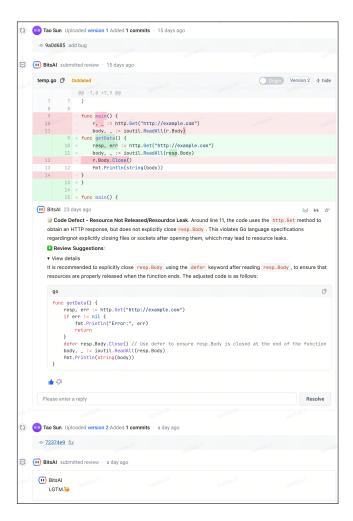

Figure 5: The MR interface shows the "outdated" status of the initial review comments and BɪᴛsAI-CR's final approval ("LGTM" - Looks Good To Me), indicating that the developer successfully resolves issues through next commit.

the code waiting to be merged, confirms the review category, pinpoints problematic code lines, and provides relevant comments. After developers address these review comments and make the necessary code modifications, BɪᴛsAI-CR re-evaluates the changes and, as shown in Figure 5, marks the original review comments as "outdated" and provides an "LGTM" (Looks Good To Me) approval, indicating that the code modifications successfully addressed the identified issues and now meets the required quality standards.

4 Experiments and Analysis

This section presents a comprehensive evaluation of BɪᴛsAI-CR through both offline experiments and large-scale industrial deployment. We first detail our model training approach and configuration, followed by a rigorous offline evaluation of code review capabilities using a carefully curated dataset. We then conduct an ablation study to validate the necessity of ReviewFilter. The analysis extends to online performance metrics in production environments, including

precision trends and user retention rates. Throughout these experiments, we demonstrate BɪᴛsAI-CR's effectiveness across multiple dimensions of code review, from technical precision to practical utility in real-world development scenarios.

4.1 Model Training

The development of BɪᴛsAI-CR is primarily driven by enterprise security requirements and data privacy considerations. As a result, we utilize `Doubao-Pro-32K-0828`, ByteDance's developed LLM, which ensures compliance with our security policies while maintaining high-performance standards. It's one of the most advanced LLMs and has a rich knowledge base and robust analytical capabilities.

We employ a fine-tuning approach using the Low-Rank Adaptation (LoRA)[13] technique on the `Doubao-Pro-32K-0828` for both RuleChecker and ReviewFilter. Based on our analysis of historical review data, which shows that 99% of review samples contain fewer than 8192 tokens, we use `Doubao-Pro-32K-0828` with an 8192 sequence length. We fine-tune it for 5 epochs with a batch size of 8. The LoRA configuration includes a rank of 128 and an alpha of 256. We use a learning rate of 0.00005 with gradient accumulation over 1 step and a warmup step rate of 0.05 to stabilize early training.

4.2 Evaluation of Code Review Capabilities

To evaluate BɪᴛsAI-CR's effectiveness in code review, we collect an offline dataset consisting of 1397 cases sampled from the production codebase, where 767 samples violate and 630 samples follow the code best practices. These cases are drawn from the taxonomy of review rules, as categorized in Table 1.

Following the LLM-as-a-judge methodology[52], we employ `Doubao-Pro-32K-0828` to evaluate automatically our business code dataset while preserving data confidentiality. The review comment is deemed correct only if the model determines it aligns with the ground truth. The experimental results are presented in Table 2, where we compare our approach against strong baseline models including Qwen2.5-Coder-32b-instruct[15] and Deepseek-v2.5[54]. We evaluated two versions: (1) a base version (BɪᴛsAI-CR w/o Taxonomy) trained on randomly sampled human internal review data without taxonomic classification (similar to existing open-source approaches[21]), and (2) a taxonomy-guided version (BɪᴛsAI-CR) where the training data is specifically constructed according to our taxonomy of review rules while maintaining the same data volume. The results demonstrate that while our base BɪᴛsAI-CR already outperforms baseline models, the taxonomy-guided version achieves substantially higher precision across all categories, reaching 57.03% overall precision compared to the base model's 16.83%. These results demonstrate that our taxonomy enhances review precision.

Ablation Study for ReviewFilter. To evaluate the necessity of the ReviewFilter component, we conduct an ablation study comparing model performance with and without this filtering stage, as shown in Table 2. Our experiments reveal that advanced LLMs' raw outputs, even with extensive fine-tuning, fail to meet the precision requirements for production deployment. This limitation manifests in persistent hallucinations across various code review scenarios. For example, we observed that the fine-tuned RuleCheck module often incorrectly identifies formatting issues. In one case, the output

Model	Review Dimension	Only RuleChecker		With ReviewFilter	
		Precision (%)	Recall (%)	Precision (%)	Recall (%)
Qwen2.5-Coder-32b-instruct	ALL	10.14	21.90	10.62	20.86
Deepseek-v2.5		9.27	16.30	9.50	16.17
Doubao-Pro-32K-0828		7.65	13.56	7.70	13.56
BɪᴛsAI-CR w/o Taxonomy		16.83	31.55	30.92	22.29
BɪᴛsAI-CR		**57.03**	45.50	**65.59**	39.77
Qwen2.5-Coder-32b-instruct	Security Vulnerability	18.52	35.71	20.83	35.71
Deepseek-v2.5		14.29	23.81	14.49	23.81
Doubao-Pro-32K-0828		9.86	16.67	10.00	16.67
BɪᴛsAI-CR w/o Taxonomy		10.00	16.67	19.23	11.90
BɪᴛsAI-CR		**58.82**	47.62	**61.29**	45.24
Qwen2.5-Coder-32b-instruct	Code Defect	7.62	16.89	7.92	16.23
Deepseek-v2.5		8.61	16.23	8.96	16.23
Doubao-Pro-32K-0828		4.72	8.94	4.75	8.94
BɪᴛsAI-CR w/o Taxonomy		16.20	34.44	26.19	21.85
BɪᴛsAI-CR		**53.88**	43.71	**68.92**	33.77
Qwen2.5-Coder-32b-instruct	Maintainability and Readability	11.06	24.54	11.56	22.98
Deepseek-v2.5		8.97	15.40	9.11	15.14
Doubao-Pro-32K-0828		9.92	17.23	9.97	17.23
BɪᴛsAI-CR w/o Taxonomy		18.15	31.85	35.94	24.02
BɪᴛsAI-CR		**58.12**	46.74	**63.60**	43.34
Qwen2.5-Coder-32b-instruct	Performance Issue	14.29	20.00	14.81	20.00
Deepseek-v2.5		13.46	17.50	13.46	17.50
Doubao-Pro-32K-0828		7.69	10.00	7.84	10.00
BɪᴛsAI-CR w/o Taxonomy		16.67	22.50	42.11	20.00
BɪᴛsAI-CR		**72.00**	45.00	**72.00**	45.00

Table 2: Performance on Code Review Evalution

Reasoning Pattern	Precision (%)	Recall (%)	Filter Rate (%)	Inference Time (s/sample)
Direct Conclusion	63.27	77.50	38.75	**1.7**
Reasoning-First	65.80	**81.80**	30.00	31.0
Conclusion-First	**77.09**	69.00	**55.25**	1.7

Table 3: Performance Comparison of Reasoning Patterns

of message is like: "The variable name 'basicInfoInstantHome' does not comply with coding standards, the underscore '_' should be removed, changing it to 'basicInfoInstantHome'." This message is evidently a hallucination since the variable name 'basicInfoInstantHome' already complies with the coding standards and does not contain any underscores. In another case, it mistakes the function name 'WhenAwemeStartFrom2580' as containing a magic number, demonstrating how fine-tuned LLMs can misinterpret code context without proper filtering. The ablation results show that the addition of ReviewFilter consistently improves precision across all categories - increasing overall precision from 54.50% to 67.12% in the taxonomy-guided model. These quantitative improvements, combined with the qualitative examples of prevented hallucinations, demonstrate that the ReviewFilter is an essential component for achieving production-grade reliability in automated code review.

Reasoning Patterns of ReviewFilter. To optimize our comment validation mechanism in ReviewFilter, we evaluate three reasoning patterns described in Section 3.3. We collect an additional 400 related data for the experiment. As indicated in Table 3, the Direct Conclusion pattern records the lowest precision. Significantly, enriching the training dataset with a reasoning process can enhance the performance of the ReviewFilter. A comparative analysis of the
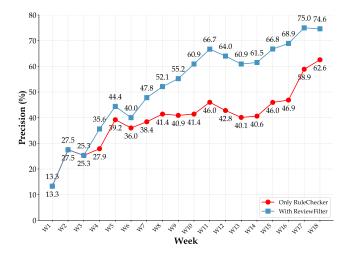
Figure 6: Weekly Progression of BɪᴛsAI-CR Precision

Figure 7: Weekly Progression of Outdated Rates and Review Rule Counts in Go Language

reasoning patterns reveals that the Reasoning-First pattern attains a superior recall of 81.80%, albeit at the cost of a noticeably prolonged inference time of 31s/sample, rendering it less feasible for deployment in production scenarios. Conversely, the Conclusion-First pattern, while obtaining the lowest recall rate at 69.00% and the highest filter rate, measured as the ratio of comments identified as erroneous by ReviewFilter, of 55.25%. This pattern notably improves the precision to 77.09%. Furthermore, the inference time for the Conclusion-First pattern is comparably short, akin to that of the Direct Conclusion pattern, due to its dependence predominantly on the initial token generation during inference. Given the prioritization of precision over recall in code review contexts, the Conclusion-First pattern offers a balance between effectiveness and operational efficiency, thereby making it the preferred choice.

4.3 Online Product Performance

To illustrate the effectiveness of our data flywheel approach, we track the performance of BɪᴛsAI-CR over time.

Precision Trends. Figure 6 illustrates the precision trends of BɪᴛsAI-CR over an 18-week period. In the initial three weeks, without the taxonomy of review rules method and two-stage approach, the overall precision remained stagnant at around 25% (with no distinction between RuleChecker and ReviewFilter at this stage). After implementing the taxonomy of review rules method combined with the two-stage approach, both components began to show gradual improvement. The RuleChecker's precision increased from an initial 27.9% to 62.6%, while the ReviewFilter's precision rose from 35.6% to a peak of 75%. These improvements demonstrate the sustainable iterative nature of our approach. Notably, figure 6 shows that RuleChecker's precision consistently remains lower than ReviewFilter's precision, enabling the ReviewFilter to deliver superior results to users and thereby validating its essential role.

Outdated Rate Trends. The effectiveness of our approach is demonstrated in Figure 7, which tracks three key metrics over an 18-week period only in Go Language. We select Go because it is a primary language at ByteDance, which establishes support and optimization
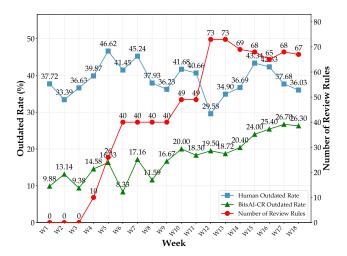
processes from early code reviews, making it ideal for observing our methodology's effect. During the first 10 weeks, despite the continuous increase in review rules, the Outdated Rate remains relatively stable at around 15%, even as the precision metrics show improvement. After expanding our review rules to 73, we observe a significant increase in the Outdated Rate, reaching approximately 20%, coinciding with a peak precision rate of 63%. Following this period, the Outdated Rate stabilizes temporarily. Starting from week 14, we begin optimizing BɪᴛsAI-CR by removing underperforming review rules based on Outdated Rate and precision metrics, which leads to a gradual increase in the Outdated Rate, ultimately reaching a peak of 26.7% by week 18. For comparison, the Human Outdated Rate at ByteDance representing how often code flagged by human reviewers gets modified, fluctuates between 35%–46%, serving as a baseline for effective code review impact. The gradual convergence of BɪᴛsAI-CR's Outdated Rate toward human-level performance demonstrates the effectiveness of our data flywheel.

User Feedback and Expert Interview. To validate the effectiveness of BɪᴛsAI-CR, we conducted a comprehensive evaluation combining quantitative surveys (N=137) and in-depth expert interviews (N=12). The survey participants were randomly selected across different programming languages, with Go (50%), frontend technologies (25%), and other languages including Java and Python (25%) represented. The results showed strong positive reception, with 74.5% (102/137) of users affirming the value and effectiveness of BɪᴛsAI-CR's code reviews. Among the remaining 25.5% (35/137) who suggested improvements, the feedback was categorized as follows: incorrect comments (10.9%, 15/137), correct but unnecessary comments (12.4%, 17/137), irrelevant feedback (1.5%, 2/137), and comprehension difficulties (0.7%, 1/137). To complement the survey data, we conducted structured 10-minute interviews with 12 expert developers who had used BɪᴛsAI-CR for more than one month. The participants represented diverse roles including frontend, backend, and algorithm development. Table 4 summarizes the key findings

Table 4: Analysis of Expert Interview Feedback (N=12)

Aspect	Key Findings
Quality Assessment	Unanimous agreement on \method{} utility (12/12) Quality improvement suggestions (3/12) The Go feels good, other languages need to be improved (4/12)
Performance Concerns	Feedback on latency of review generation (7/12) Strong desire for improved processing speed (9/12)
Feature Requests	Support for customizable review rules (11/12) Don't review the code auto-generated by the framework (2/12) More program languages review support (2/12) One-click suggestion application functionality (11/12) SDK provision for pre-push review integration (1/12)

Figure 8: Weekly User Retention Rate of BɪᴛsAI-CR

from these interviews, categorized by usage patterns, quality assessment, performance considerations, and feature requests.

4.4 Large-Scale Industrial Deployment

Before full-scale implementation, we conduct canary testing to validate the system's performance and user acceptance. This phased deployment approach ensures minimal disruption to users and also allows us to rapidly collect user feedback data and make immediate adjustments based on online responses.

Currently,BɪᴛsAI-CR fully deploys across ByteDance's development teams. It boasts over **12k Weekly Active Users (WAU)** and **210k Weekly Page Views (WPV)**, demonstrating its widespread adoption and effective integration into the development workflow. Furthermore, as shown in Figure 8, the system shows a second-week retention rate of 61.64%, and can still maintain a high retention rate of around 48% through the eight weeks. To our knowledge, this represents the first documented retention rate benchmark for large-scale code intelligence tools, providing valuable reference metrics for other organizations implementing similar systems.

5 Lessons Learned and Practical Insights

The large-scale implementation of BɪᴛsAI-CR has yielded valuable insights that can guide other organizations in developing and deploying automated code review systems. We present our key findings across the following critical dimensions:

Taxonomy of Review Rules ☛ *enables systematic code issue categorization, data collection, and performance evaluation.*

Our experience demonstrates that a well-defined taxonomy of code review rules serves as the cornerstone for building effective automated review systems. The taxonomy (Table 1) provides crucial benefits: (1) it enables systematic issue categorization and detection, guiding the development of BɪᴛsAI-CR which improved precision from 30.92% to 65.59% with ReviewFilter in the Go language, (2) it facilitates structured data collection and labeling, ensuring consistent training data quality, and (3) it provides clear criteria for evaluating and improving system performance across different review dimensions. Unlike traditional approaches that lack systematic data-driven evolution, our taxonomy-driven approach enables more precise and actionable code review through structured issue classification and targeted optimization.

Two-Stage Review Generation ☛ *enhances automated code review reliability by validating identified issues.*

Our findings demonstrate that a two-stage review generation approach, combining RuleChecker and ReviewFilter, is crucial for achieving production-grade reliability in automated code review. While LLMs can identify potential issues, their tendency to produce false positives and hallucinations necessitates a robust validation mechanism. Our implementation of ReviewFilter as a dedicated validation module significantly improved the BɪᴛsAI-CR's reliability, increasing overall precision from 60% to 75%. This improvement demonstrates that a dedicated filter module is a critical component for building dependable automated code review systems.

Precision and Outdated Rate Metrics ☛ *guides data flywheel optimization through user-centric evaluation.*

Our deployment track record demonstrates that prioritizing precision over recall metrics is crucial for successful automated code review adoption. High precision builds user trust early without disturbing users or causing them to abandon utilizing BɪᴛsAI-CR, encouraging continued system utilization and enabling iterative improvements. However, we discover that traditional precision metrics alone are insufficient, as they require unsustainable manual labeling and fail to capture user acceptance of review rules. To address these limitations, we introduce the Outdated Rate metric as a complementary measure that reflects user acceptance of review suggestions. The combination of precision and Outdated Rate metrics proves instrumental in driving our data flywheel, facilitating large-scale deployment, and achieving positive user feedback through continuous system refinement.

6 Related Work

Code Large Language Models. Code LLMs [10, 14, 53] represent a crucial vertical within the broader LLM domain. These models advance software engineering by enhancing code intelligence[51]. Recent models, including GPT-4[30], Llama3[6], Qwen2.5-Coder[15],

and DeepSeek-V2.5[54], demonstrate significant capabilities in multilingual code generation and debugging tasks. Code LLMs transform various aspects of software development, spanning code generation [3, 36, 42, 43], program repair [23, 37], log parsing [20], web design [41, 45], and other applications [24, 46].

LLM-Based Code Review. LLM applications in code review represent an active research area that encompasses both implementation and evaluation approaches. Foundational work establishes deep learning for code review, with studies like CodeReviewer [21] and related research [4, 19, 39, 49] exploring pre-trained models for review tasks. AutoCommenter [40] advances this field by analysing code compliance with best practices, though it limits output to Google's best practices URLs without specific code suggestions. LLaMA-Reviewer [25] marks a significant advance through LLAMA model fine-tuning for code review. Recent research investigates hyperparameters, fine-tuning strategies, and prompt engineering [9, 11, 31, 50], while studies on multi-agent systems [33, 38] expand automated review capabilities. Parallel evaluation efforts include EvaCRC [47], which introduces a framework for assessing review comments. Additional studies [1, 5, 18, 29] explore review quality assessment and human-AI consistency. Notable findings indicate LLMs can surpass human performance in error detection [27], while research also addresses cognitive biases in review processes [16].

7 Conclusion and Feature Work

This paper presents BitsAI-CR, a comprehensive automated code review system that tackles both the efficiency bottlenecks in enterprise-scale review processes and the fundamental limitations of existing LLM-based solutions, particularly their insufficient comment effectiveness and lack of systematic improvement mechanisms. To achieve this, we introduce a novel taxonomy of review rules that serve as the foundation for our two-stage approach: RuleChecker for initial issue detection and ReviewFilter for precision enhancement. We further propose the Outdated Rate metric to evaluate comment practicality and drive systematic improvements through a data flywheel mechanism. Our empirical evaluation demonstrates the effectiveness of this approach, achieving an acceptable precision rate in comment generation and a competitive Outdated Rate in Go language reviews. The successful deployment at ByteDance validates its scalability and practical value in enterprise-scale software development environments.

Our future work will focus on these key directions for technical and service enhancement: First, we plan to expand language coverage from our current support of five mainstream programming languages to comprehensive coverage of all programming languages. Second, we will enhance our review rules, which currently focus primarily on function-level understanding with limited contextual information, by developing effective cross-file review capabilities. Through these continuous efforts, we strive to enrich and strengthen our system to provide more comprehensive and in-depth code review services, ultimately helping developers improve both code quality and engineering efficiency.

References

[1] Toufique Ahmed, Premkumar Devanbu, Christoph Treude, and Michael Pradel. 2024. *Can LLMs Replace Manual Annotation of Software Engineering Artifacts?* arXiv.org. https://arxiv.org/abs/2408.05534v1

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 712–721.

[3] Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, et al. 2024. McEval: Massively Multilingual Code Evaluation. *arXiv preprint arXiv:2406.07436* (2024).

[4] Qiuyuan Chen, Dezhen Kong, Lingfeng Bao, Chenxing Sun, Xin Xia, and Shanping Li. 2022. Code Reviewer Recommendation in Tencent: Practice, Challenge, and Direction*. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (Pittsburgh, PA, USA, 2022-05). IEEE, 115–124. doi:10.1109/ICSE-SEIP55303.2022.9794124

[5] Lee Dong-Kyu. 2024. *A GPT-based Code Review System for Programming Language Learning.* arXiv. https://arxiv.org/abs/2407.04722v1

[6] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[7] Sigrid Eldh. 2024. Code Review Evolution. *IEEE Software* 41, 5 (2024), 4–8.

[8] Pär Emanuelsson and Ulf Nilsson. 2008. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science* 217 (2008), 5–21.

[9] Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. 2024. *Exploring the Capabilities of LLMs for Code Change Related Tasks.* arXiv.org. https://arxiv.org/abs/2407.02824v1

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[11] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024-02-06) *(ICSE '24)*. Association for Computing Machinery, 1–13. doi:10.1145/3597503.3623306

[12] Md Asif Haider, Ayesha Binte Mostofa, Sk Sabit Bin Mosaddek, Anindya Iqbal, and Toufique Ahmed. 2024. Prompting and Fine-tuning Large Language Models for Automated Code Review Comment Generation. *arXiv preprint arXiv:2411.10129* (2024).

[13] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[14] Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, et al. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905* (2024).

[15] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[16] Tobias Jetzen, Xavier Devroey, Nicolas Matton, and Benoît Vanderose. 2024. *Towards Debiasing Code Review Support.* doi:10.48550/arXiv.2407.01407 arXiv:2407.01407 [cs]

[17] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*. 92–101.

[18] Charles Koutcheme, Nicola Dainese, Arto Hellas, Sami Sarsa, Juho Leinonen, Syed Ashraf, and Paul Denny. 2024. *Evaluating Language Models for Generating and Judging Programming Feedback.* arXiv.org. https://arxiv.org/abs/2407.04873v1

[19] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. 2022. *AUGER: Automatically Generating Review Comments with Pre-training Models.* doi:10.48550/arXiv.2208.08014 arXiv:2208.08014 [cs]

[20] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, Lionel C Briand, and Michael R Lyu. 2024. Exploring the Effectiveness of LLMs in Automated Logging Statement Generation: An Empirical Study. *IEEE Transactions on Software Engineering* (2024).

[21] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. *Automating Code Review Activities by Large-Scale Pre-training.* doi:10.48550/arXiv.2203.09095 arXiv:2203.09095 [cs]

[22] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977* (2024).

[23] Shukai Liu, Linzheng Chai, Jian Yang, Jiajun Shi, He Zhu, Liran Wang, Ke Jin, Wei Zhang, Hualei Zhu, Shuyue Guo, et al. 2024. Mdeval: Massively multilingual code debugging. *arXiv preprint arXiv:2411.02310* (2024).

[24] Siyao Liu, He Zhu, Jerry Liu, Shulin Xin, Aoyan Li, Rui Long, Li Chen, Jack Yang, Jinxiang Xia, ZY Peng, et al. 2024. FullStack Bench: Evaluating LLMs as Full Stack Coder. *arXiv preprint arXiv:2412.00535* (2024).

[25] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. *LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning*. arXiv.org. https://arxiv.org/abs/2308.11148v2

[26] Srijoni Majumdar, Ayush Bansal, Partha Pratim Das, Paul D Clough, Kausik Datta, and Soumya Kanti Ghosh. 2022. Automated evaluation of comments to aid software maintenance. *Journal of Software: Evolution and Process* 34, 7 (2022), e2463.

[27] Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. 2024. *LLM Critics Help Catch LLM Bugs*. arXiv.org. https://arxiv.org/abs/2407.00215v1

[28] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21 (2016), 2146–2189.

[29] Atharva Naik, Marcus Alenius, Daniel Fried, and Carolyn Rose. 2024. *CRScore: Grounding Automated Evaluation of Code Review Comments in Code Claims and Smells*. doi:10.48550/arXiv.2409.19801 arXiv:2409.19801 [cs]

[30] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023). https://arxiv.org/abs/2303.08774

[31] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2024. *Fine-Tuning and Prompt Engineering for Large Language Models-based Code Review Automation*. arXiv.org. https://arxiv.org/abs/2402.00905v4

[32] Rachel Potvin and Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. *Commun. ACM* 59, 7 (2016), 78–87.

[33] Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. 2024. *AI-powered Code Review with LLMs: Early Results*. arXiv.org. https://arxiv.org/abs/2404.18496v1

[34] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 181–190.

[35] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T Stolee, and Brittany Johnson. 2017. Evaluating how static analysis tools can reduce code review effort. In *2017 IEEE symposium on visual languages and human-centric computing (VL/HCC)*. IEEE, 101–105.

[36] Tao Sun, Linzheng Chai, Jian Yang, Yuwei Yin, Hongcheng Guo, Jiaheng Liu, Bing Wang, Liqun Yang, and Zhoujun Li. 2024. Unicoder: Scaling code large language model via universal code. *arXiv preprint arXiv:2406.16441* (2024).

[37] Tao Sun, Yang Yang, Xianfu Cheng, Jian Yang, Yintong Huo, Zhuoren Ye, Rubing Yang, Xiangyuan Guan, Wei Zhang, Hangyuan Ji, et al. [n. d.]. RepoFixEval: A Repository-Level Program Repair Benchmark From Issue Discovering to Bug Fixing. ([n. d.]).

[38] Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawende F. Bissyande. 2024. *CodeAgent: Autonomous Communicative Agents for Code Review*. arXiv.org. https://arxiv.org/abs/2402.02172v5

[39] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. *Using Pre-Trained Models to Boost Code Review Automation*. doi:10.48550/arXiv.2201.06850 arXiv:2201.06850 [cs]

[40] Manushree Vijayvergiya, Małgorzata Salawa, Ivan Budiselić, Dan Zheng, Pascal Lamblin, Marko Ivanković, Juanjo Carin, Mateusz Lewko, Jovan Andonov, Goran Petrović, Daniel Tarlow, Petros Maniatis, and René Just. 2024. *AI-Assisted Assessment of Coding Practices in Modern Code Review*. arXiv.org. doi:10.1145/3664646.3665664

[41] Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael R Lyu. 2024. Automatically generating UI code from screenshot: A divide-and-conquer-based approach. *arXiv preprint arXiv:2406.16386* (2024).

[42] Chaozheng Wang, Shuzheng Gao, Cuiyun Gao, Wenxuan Wang, Chun Yong Chong, Shan Gao, and Michael R Lyu. 2024. A Systematic Evaluation of Large Code Models in API Suggestion: When, Which, and How. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 281–293.

[43] Chaozheng Wang, Zongjie Li, Cuiyun Gao, Wenxuan Wang, Ting Peng, Hailiang Huang, Yuetang Deng, Shuai Wang, and Michael R Lyu. 2024. Exploring Multi-Lingual Bias of Large Code Models in Code Generation. *arXiv preprint arXiv:2404.19368* (2024).

[44] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[45] Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zhiyao Xu, and Michael R Lyu. 2024. Interaction2Code: How Far Are We From Automatic Interactive Webpage Generation? *arXiv preprint arXiv:2411.03292* (2024).

[46] Jian Yang, Jiaxi Yang, Ke Jin, Yibo Miao, Lei Zhang, Liqun Yang, Zeyu Cui, Yichang Zhang, Binyuan Hui, and Junyang Lin. 2024. Evaluating and Aligning CodeLLMs on Human Preference. *arXiv preprint arXiv:2412.05210* (2024).

[47] Lanxin Yang, Jinwei Xu, Yifan Zhang, He Zhang, and Alberto Bacchelli. 2023. EvaCRC: Evaluating Code Review Comments. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco CA USA, 2023-11-30). ACM, 275–287.

[48] Zezhou Yang, Cuiyun Gao, Zhaoqiang Guo, Zhenhao Li, Kui Liu, Xin Xia, and Yuming Zhou. 2024. *A Survey on Modern Code Review: Progresses, Challenges and Opportunities*. arXiv.org. https://arxiv.org/abs/2405.18216v1

[49] Ying Yin, Yuhai Zhao, Yiming Sun, and Chen Chen. 2023. Automatic Code Review by Learning the Structure Information of Code Graph. 23, 5 (2023), 2551. Issue 5. doi:10.3390/s23052551

[50] Yongda Yu, Guoping Rong, Haifeng Shen, He Zhang, Dong Shao, Min Wang, Zhao Wei, Yong Xu, and Juhong Wang. 2024. Fine-Tuning Large Language Models to Improve Accuracy and Comprehensibility of Automated Code Review. (2024), 3695993. doi:10.1145/3695993

[51] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989* (2023).

[52] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems* 36 (2023), 46595–46623.

[53] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658* (2024).

[54] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).