Journal of Information System Research (JOSH)
Volume 7, No. 4, Juli 2026, pp 1174−1183
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i4.10618

Automating Clean Architecture Conformance via AST-Based Code Smell Detection and Auto-Refactoring in Laravel Projects

Ahmad Fatih Hibatillah, Fitri Wulandari*

Department of Informatics, Faculty of Science and Technology, Universitas Islam Negeri Sunan Kalijaga Yogyakarta, Yogyakarta
Jl. Laksda Adisucipto, Papringan, Caturtunggal, Kec. Depok, Kabupaten Sleman, Daerah Istimewa Yogyakarta, Indonesia
Email: [1]22106050032@student.uin-suka.ac.id, [2,*]fitri.wulandari@uin-suka.ac.id
Correspondence Author Email: fitri.wulandari@uin-suka.ac.id
Submitted: **05/07/2026;** Accepted: **20/07/2026;** Published: **21/07/2026**

Abstract−While MVC frameworks like Laravel accelerate development, their inherent architectural flexibility frequently induces structural code smells such as fat controllers and direct database access that inevitably lead to architectural erosion and technical debt. This research proposes a Visual Studio Code (VS Code) extension designed to automate Clean Architecture conformance through real-time Abstract Syntax Tree (AST) parsing. Utilizing the Tree-sitter parsing system, the extension continuously evaluates the codebase to detect architectural violations. Beyond passive diagnostics, the tool facilitates interactive auto-refactoring. To ensure business logic integrity during migration, the refactoring engine employs deterministic AST node transformations that safely isolate data access operations into dedicated service layers without altering operational behavior. Evaluation through Black-Box Testing confirmed high detection accuracy across predefined architectural anti-patterns. Furthermore, User Acceptance Testing (UAT) conducted with industry experts yielded an average acceptance score of 4.18 out of 5.00. Ultimately, this AST-based approach provides a pragmatic solution to actively prevent architectural decay, mitigate technical debt, and significantly enhance the long-term maintainability of Laravel-based projects. The main contribution of this research is the transition of architectural code smell mitigation from a passive diagnostic reporting mechanism into an active, real-time, AST-driven auto-refactoring tool directly integrated within the developer's workspace.

Keywords: Auto-Refactoring; Clean Architecture; Code Smell; Laravel; Visual Studio Code Extension

1. INTRODUCTION

Modern software engineering dictates that the development of applications must prioritize functional code structures that guarantee high testability and long-term maintainability. The software maintenance phase frequently consumes the absolute largest portion of project resources. Empirical evidence demonstrates that software maintenance can dominate 70% to 90% of the total overall development lifecycle costs [1]. This massive cost inflation fundamentally roots from the negligence of code quality standards during the initial rapid project phases. Poor coding practices directly create a phenomenon known as technical debt. Technical debt represents the consequences of rapid but sub-optimal implementation decisions that act like financial debt, accumulating interest over time in the form of maintenance burdens [2]. Quantitative studies of proprietary production codebases reveal that this technical debt causes developers to lose up to 42% of their productive time merely untangling and fixing messy logic instead of building new features [3].

To achieve production efficiency and accelerate delivery timelines, developers widely utilize modern web frameworks such as Laravel [4], particularly within the PHP server-side scripting ecosystem [5]. However, utilizing a framework does not automatically guarantee a high-quality code structure. Empirical studies observing various Laravel-based projects prove that code quality violations including severe code smells, bugs, and significant logic duplication still occur frequently in real-world implementations [6].

The root of these continuous quality degradation issues often lies in the built-in architectural flexibility of the framework itself. Although Laravel formally adopts the Model-View-Controller (MVC) architectural pattern to facilitate the separation of concerns, its practical implementation does not provide strict, enforced boundaries on the distribution of business logic [7]. The pure MVC pattern leaves the regulation of functional flow entirely to the discipline of the developers, frequently leading to sub-optimal code organization that deviates from the original design [8].

This absence of strict, compiler-level boundaries opens critical loopholes for excessive logic accumulation in a single controller component. Consequently, controllers transition into massive components handling application flow, data validation, and complex business rules simultaneously, a structural anti-pattern widely recognized as a "fat controller" [9]. Furthermore, the availability of highly accessible instant features within the framework, such as the Eloquent Object-Relational Mapper (ORM), unintentionally encourages poor architectural practices. The instant accessibility to data manipulation tempts developers to practice direct database access from the presentation layer [7]. By doing so, developers bypass essential intermediary abstraction layers, such as the service layer or the repository layer. This lack of strict boundaries creates severe tight coupling between the presentation mechanism and the data infrastructure layer, accelerating technical debt and hindering testability [2].

Bad coding practices are tangible manifestations of "code smells," which are surface indicators corresponding to deeper, more systemic problems in software design [10]. In the specific context of PHP-based web applications, the presence of code smells correlates strongly with decreased maintainability, higher defect density, and an increased risk of software vulnerabilities [9]. Code smells are classified into several categories,

Journal of Information System Research (JOSH)
Volume 7, No. 4, Juli 2026, pp 1174−1183
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i4.10618

including bloaters (e.g., large classes or long methods), object-oriented abusers, change preventers, dispensables, and couplers [10]. In Laravel projects, the most destructive smells manifest as fat controllers, direct database access, and high logic complexity. High complexity severely impacts the mental burden of developers during debugging and testing. Traditional metrics like Cyclomatic Complexity often fall short as they ignore structural nesting; thus, modern evaluations utilize Cognitive Complexity to qualitatively measure readability and developer cognitive load [11]. If left unresolved, these code smells trigger architectural erosion a detrimental state where the actual code implementation progressively deviates and decays from the initially established architectural design [2].

To overcome code smells and proactively prevent architectural erosion, developers require continuous Architecture Conformance Checking (ACC) [12]. This is achieved through strict adherence to robust design guidelines, such as Clean Architecture [13]. Introduced by Robert C. Martin, Clean Architecture emphasizes a strict separation of concerns, isolating core enterprise business rules entirely from technical details like web frameworks, user interfaces, and databases. The most fundamental principle in this architecture is the Dependency Rule, which dictates that source code dependencies must strictly point inward toward the core business logic [14]. If applied consistently, this principle prevents structural code smells such as fat controllers and direct database access early on by forcing data access and business rules into isolated layers.

Various approaches have been developed to detect code smells automatically, notably utilizing the Abstract Syntax Tree (AST) hierarchy structure [15]. AST breaks down source code into discrete nodes representing programming language constructs, ignoring irrelevant elements like spaces and formatting to generate a context-rich structural representation [16]. This hierarchical approach is proven to be far more effective, precise, and capable of understanding code relations contextually compared to mere text-based pattern searches using Regular Expressions (Regex), which lack semantic awareness and produce high false-positive rates [17]. Modern Integrated Development Environments (IDEs) provide integrated facilities to support such advanced analyses [18], particularly through static code analysis mechanisms [19]. Visual Studio Code (VS Code), a highly modular IDE, supports this through its Extension API, which grants access to workspace monitoring, diagnostic collections, and workspace edit capabilities [20]. To ensure reliability in such complex tooling, development often leverages strictly typed languages like TypeScript [21].

Several previous studies have explored code smell detection and software architecture maintenance. For instance, Wadi et al. [15] utilized AST for code smell detection, but their scope was limited to general syntax without addressing architectural layer boundaries. Similarly, Nuraminah and Ammar [16] demonstrated the high precision of AST, though their application was strictly for code plagiarism detection rather than architectural conformance.

Furthermore, Alharbi and Alshayeb [22] evaluated various automated refactoring tools, revealing that most existing tools function predominantly as passive diagnostic reporters instead of active structural modifiers. In the context of architectural improvement, Nugroho et al. [23] successfully applied Clean Architecture to existing applications; however, the refactoring process was executed manually, which is time-consuming and prone to human error. The research gap is evident: while previous studies have successfully implemented AST for structural analysis and recognized the importance of Clean Architecture, there is a critical absence of an integrated tool that combines real-time AST parsing with automated, rule-based refactoring specifically targeting Laravel presentation layer boundaries. Addressing these critical research gaps, this research aims to design and build a Visual Studio Code Extension capable of accurately detecting code smells using Clean Architecture principles as its structural evaluation standard. This extension leverages Tree-sitter, an incremental parsing system designed for high-speed AST generation with robust error recovery [24].

By executing efficiently within a WebAssembly (WASM) environment, it prevents main thread blocking and ensures real-time responsiveness [25]. Crucially, the system goes beyond passive detection by acting as an active real-time assistant, facilitating interactive auto-refactoring executions. By automating the extraction of misplaced logic from the presentation layer to the appropriate service boundaries, this extension ensures the application of Clean Architecture principles remains consistent, seamlessly preventing architectural erosion during the software development cycle. Therefore, the main contributions of this research are twofold: (1) theoretically, it introduces a real-time, active intervention model for Architecture Conformance Checking (ACC) using AST and WebAssembly; and (2) practically, it delivers a functional VS Code extension capable of executing deterministic auto-refactoring to enforce Clean Architecture boundaries in Laravel projects.

2. RESEARCH METHODOLOGY

2.1 Research Object

The primary object of this research comprises PHP source code files specifically residing within the presentation layer of the Laravel framework, namely the app/Http/Controllers directory. The analysis exclusively targets architectural deviations within these controller components, treating the surrounding components (e.g., Models, Services, and Repositories) as external contextual boundaries for the Clean Architecture evaluation.

Journal of Information System Research (JOSH)
Volume 7, No. 4, Juli 2026, pp 1174−1183
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i4.10618

2.2 Research Stages and Flowchart

The software engineering methodology utilized in the design and development of this Visual Studio Code Extension is Extreme Programming (XP) [26]. This agile methodology was strategically selected because its highly iterative and incremental approach aggressively prioritizes technical quality and allows for rapid adaptation to continuously changing requirements. The XP framework structurally abandons the traditional waterfall approach, replacing it with four continuous, sequential phases: Planning, Design, Coding, and Testing [27]. This rapid iteration relies heavily on continuous testing to minimize architectural modification costs if structural tuning is required mid-development [28]. Furthermore, this methodology heavily depends on continuous integration to prevent the accumulation of technical debt [29]. Consequently, this XP framework is deemed highly ideal for building precision-based analytical developer tooling [30]. The complete procedural flow of this research methodology is illustrated in the flowchart in Figure 1.

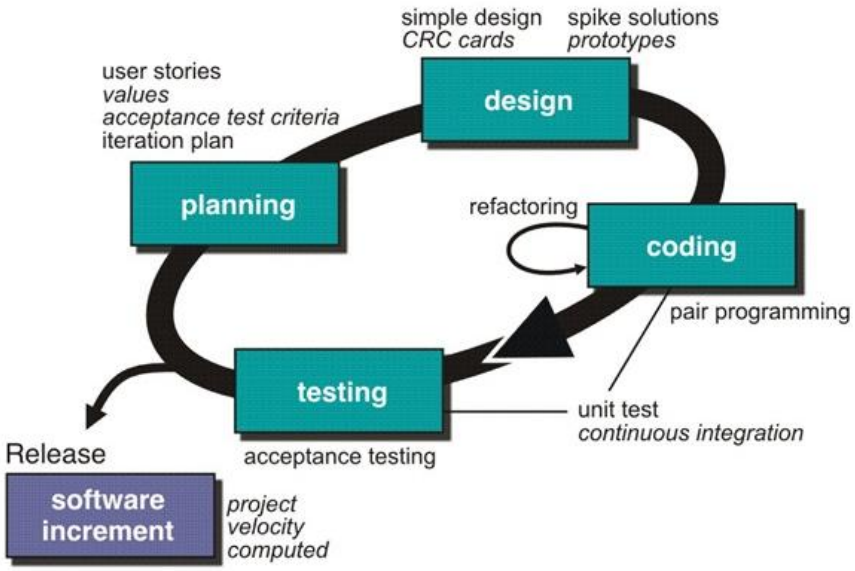

Figure 1. Flowchart of the Extreme Programming Development Methodology

The system was completely programmed using TypeScript. The research timeline was chronologically structured into two main development iterations:

2.2.1 Phase 1: Structural Code Violation Detection (First Iteration)

a) Planning: The procedure begins by defining the detection rules based on Clean Architecture principles. The targeted violations are logic accumulation in the presentation layer (fat controller), high logical complexity (high complexity), and direct database access from the controller (direct database access).
b) Design: The system architecture is designed to comprise a background file watcher, an Abstract Syntax Tree (AST) parser, a rule-based validation engine, and a diagnostic manager to project warnings onto the editor.
c) Coding: The procedure integrates tree-sitter-php compiled in WebAssembly (WASM) to mathematically construct the AST representation of the PHP code. An AST traversal algorithm is implemented to detect the three predefined violations. Specifically, the algorithm evaluates fat controllers (flagged if > 150 lines, > 7 methods, > 30 effective lines, and > 4 injected dependencies), high complexity (using cognitive complexity scoring to evaluate nested control structures), and tracks model instances to flag direct database access. Upon detection, the system generates diagnostic warnings.
d) Testing: The detection algorithm is tested in real-time to ensure diagnostic warnings appear precisely at the correct spatial coordinates of the violating code without producing false positives.

2.2.2 Phase 2: System Response and Auto-Refactoring (Second Iteration)

a) Planning: The handling strategy is defined. Fat controller and high complexity violations are handled by providing warnings and descriptive recommendations. Conversely, direct database access is targeted for automated code restructuring (auto-refactoring). Supporting managerial features are also planned: CSmell Summary (for holistic analysis reports), CSmell Config (for dynamic threshold management), and Ignore Violation (for custom user exceptions).
b) Design: The procedural design introduces a Code Action Provider to offer diagnostic-based repair options and a Workspace Edit mechanism to execute text modifications safely. The Summary Manager, Config Manager, and Ignore Manager are also architected.

Journal of Information System Research (JOSH)
Volume 7, No. 4, Juli 2026, pp 1174−1183
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i4.10618

c) Coding: The Quick Fix API is implemented to handle direct database access. The algorithmic process extracts the database logic from the controller and dynamically migrates it to the appropriate service component transactionally. For other violations, the code restricts itself to displaying warnings.
d) Testing: The Code Action mechanism is tested to guarantee that auto-refactoring executes safely without syntax corruption, and that managerial features function perfectly according to user configurations.

2.3 Evaluation and Testing Scenarios

Data acquisition for system evaluation was systematically conducted using two distinct testing methodologies. First, functional data acquisition was strictly executed through Black-Box Testing at the end of every XP phase, treating the complex internal logic of the system as an opaque black box. This method is empirically highly effective for validating end-user functionalities [31]. Second, acceptance data acquisition was conducted through User Acceptance Testing (UAT) exclusively involving 10 specialized expert users. Empirical studies on qualitative saturation confirm that data saturation in homogeneous expert groups is consistently achieved within 9 to 17 participants [32]. To further ensure methodological rigor, this sample size is scientifically justified under the principle of Information Power theory [33]. According to this theory, a smaller sample size possesses strong statistical robustness when participants hold highly specific expertise. The quantitative data collected from all respondents was then accumulated and analyzed mathematically using class interval ranges to interpret the final qualitative acceptance categories [34].

3. RESULT AND DISCUSSION

This section extensively explains the results of the system development iterations and provides a comprehensive, multi-dimensional discussion of the testing outcomes. It validates the system's technical capabilities in detecting code smells and its practical efficacy in maintaining Clean Architecture principles within real-world Laravel projects.

3.1 System Architecture and Component Interaction

To fully contextualize the detection and refactoring capabilities, it is imperative to understand the holistic architectural design of the developed extension. The system was meticulously architected using a highly modular approach, ensuring a strict separation of concerns among its internal components. The operational architecture follows a unidirectional data flow, effectively transforming raw source code into actionable architectural modifications through several distinct computational layers.

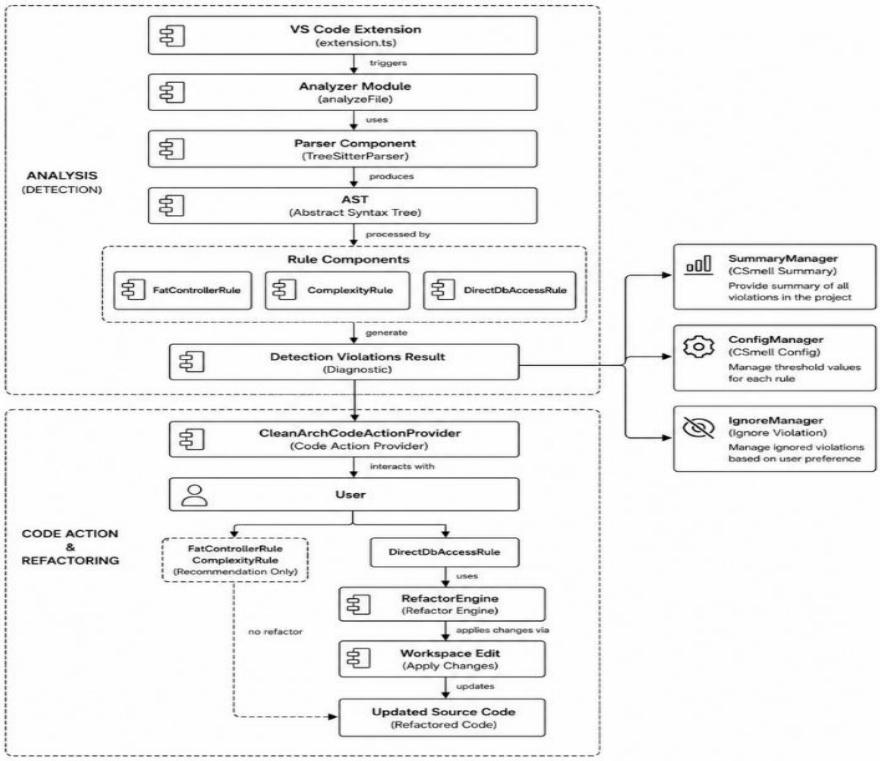

Figure 2. System Architecture and Component Interaction Diagram

The pipeline initiates at the Integration and Input Layer, which interfaces directly with the Visual Studio Code Extension API. The extension actively hooks into the editor's event listeners to monitor document lifecycle

Journal of Information System Research (JOSH)
Volume 7, No. 4, Juli 2026, pp 1174−1183
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i4.10618

changes. When a developer modifies a PHP file, this layer captures the raw string payload and instantly forwards it to the Parsing Layer. Within this layer, the WebAssembly-powered Tree-sitter engine processes the payload via incremental parsing, outputting a complete Abstract Syntax Tree (AST) representing the current structural state of the code.

Once the AST is constructed, it is passed to the Analysis Layer, which acts as the core analytical engine of the extension. The primary orchestrator here is the Analyzer Module. Adhering to the Single Responsibility Principle (SRP), the Analyzer does not compute the violations directly. Instead, it dynamically loads independent Rule Components namely the FatControllerRule, ComplexityRule, and DirectDbAccessRule and sequentially feeds them the generated AST. Each rule executes its proprietary traversal algorithm, extracting mathematical metrics and identifying architectural boundary leaks.

The findings from the Analysis Layer are systematically aggregated and pushed to the Diagnostic and Output Layer. The system encapsulates the structural flaws into standardized Diagnostic objects, complete with spatial coordinates (start and end lines) and severity levels, rendering them as visual warning squiggles within the editor's user interface.

Simultaneously, the architectural flow extends into the Action and Refactoring Layer. The CleanArchCodeActionProvider constantly monitors the active Diagnostics. When a developer interacts with a highlighted direct database access violation, the Action Provider generates a Quick Fix payload and bridges the communication to the Refactor Engine. Finally, the Refactor Engine calculates the required text mutations and executes these transformations transactionally using the Workspace Edit API. This strict, unidirectional architecture guarantees that the complex background static analysis does not interfere with the immediate safety and responsiveness of the user's primary editing workspace.

3.2 Structural Code Violation Detection

During the first development iteration, the primary focus was directed toward building the foundational static code analysis mechanism based on the Abstract Syntax Tree (AST) within the Visual Studio Code extension environment. This phase encompassed rigorous planning, architectural design, coding, and testing to ensure the system could operate seamlessly and integrally within a developer's workflow. The core objective of this iteration was to transition away from rudimentary, error-prone text-based searches and implement a hierarchical, context-aware AST parser. This approach was strategically selected to significantly enhance the accuracy of detecting code structure violations related to Clean Architecture principles specifically within the Laravel framework.

The system was meticulously designed to analyze controller files by detecting three primary violations: fat controllers, high complexity, and direct database access. This detection operates in real-time, hooking directly into the Visual Studio Code event listeners, ensuring that every code modification made by the user is instantaneously analyzed. The implementation phase emphasized a strict separation of concerns among its core components, which are detailed as follows:

1) Parser Initialization and AST Construction: The initial implementation phase focused heavily on building the parser component, which is responsible for transforming PHP source code into an Abstract Syntax Tree (AST) representation. The parser is implemented using Tree-sitter, running seamlessly within the Visual Studio Code extension environment by utilizing WebAssembly (WASM) as its low-overhead runtime. Following the initialization process, the parser becomes ready to analyze the source code via the parse function, which receives the raw source code string as input and mathematically generates the AST structure, outputting a root node. This root node serves as the absolute starting point for the traversal process.
2) Analyzer Implementation (Rules Engine): Once the construction of the AST is completed, the analyzer component functions as the system's central rules engine. This component is strictly responsible for coordinating the complex analysis process against the code structure. The analyzer is implemented as the main controller; it receives the AST root node as its primary input and subsequently distributes the analysis workload to each predefined rule. Adhering to a uniform interface pattern, each independent rule traverses the AST structure to identify specific violation patterns according to its predetermined criteria.
3) Rule Detection Implementation: The actual detection of architectural violations is executed through a sophisticated AST traversal mechanism, where each rule independently navigates the code's structural nodes. Fat Controller Rule: Evaluates four precise structural indicators: class length (> 150 lines), method count (> 7 methods, excluding the constructor), method length (> 30 effective lines within its body block, ignoring blank lines and comments), and dependency injection (> 4 dependencies injected into the class constructor). When a controller file violates these structural limits, the corresponding real-time diagnostic alert generated by the extension inside the editor to notify the developer regarding this structural bloat is illustrated in Figure 3.

Figure 3. Diagnostic Display of the Fat Controller Violation

Journal of Information System Research (JOSH)
Volume 7, No. 4, Juli 2026, pp 1174−1183
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i4.10618

High Complexity Rule: Measures the logical complexity of every method using a cognitive complexity approach. Branching (if statements) adds to the score. Loop structures (foreach, for, while) are penalized with a weight of . Choice controls (switch, try) add . Ternary operators add , and chained logical operators (&&, ||) add exactly 1 point per operator sequence. Anonymous and arrow functions are excluded. A final score greater than 5 is a warning, while a score exceeding 10 is an error. The visual warning highlight and error marker produced by the rules engine for this complex logic sequence are shown in Figure 4.

Figure 4. Diagnostic Display of the High Complexity Violation

Direct Database Access Rule: Executes in two distinct phases: Model Instance Identification (traverses assignment expressions to cache variables representing model instances) and Call Expression Analysis (flags calls utilizing the DB facade directly, invoking static methods on a model class, or manipulating data via identified instance variables). Native global functions (view, response) and common utility classes (Str, Carbon) are filtered out. The exact diagnostic squiggle and tooltip message deployed by the extension to mark this anti-pattern can be observed in Figure 5.

Figure 5. Diagnostic Display of the Direct Database Access Violation

4) Integration with Visual Studio Code (Diagnostic): Each detected violation is programmatically mapped to a diagnostic object containing precise spatial coordinates, a descriptive error message, and a calculated severity level (Error, Warning, or Information) using VS Code's DiagnosticCollection API.
5) Testing: The empirical validation of the system's real-time detection capabilities is summarized in a structured manner in Table 1.

Table 1. Black-Box Testing Results of The First Iteration

No	Testing Scenario	Expected Result	Actual Result	Conclusion
1	Controller with > 7 methods	System detects fat controller	Detected according to rule	Success
2	Controller with > 4 dependencies	System detects fat controller	Detected according to rule	Success
3	Method with > 30 effective lines	System detects fat controller	Detected according to rule	Success
4	Class with > 150 lines	System detects fat controller	Detected according to rule	Success
5	Method with complex structure (nested, loop, logical operators) with score > 10	System detects high complexity	Detected according to rule	Success
6	Direct database call (DB::, model, instance)	System detects direct database access	Detected according to rule	Success
7	Code modification in the editor	System updates detection results automatically	Diagnostic updated in real-time	Success

3.3 System Response and Auto-Refactoring

In the second iteration, development was deeply focused on enriching the system's response by implementing interactive repair recommendations (Code Actions), selective auto-refactoring mechanisms, and code quality managerial features.

Journal of Information System Research (JOSH)
Volume 7, No. 4, Juli 2026, pp 1174−1183
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i4.10618

1) Code Action Provider Implementation: Developed through the CleanArchCodeActionProvider module, this component dynamically generates Quick Fix options displayed directly on the user's editor interface when the cursor hovers over problematic lines of code.
2) Selective Response Mechanism: To maintain the integrity of application business logic, the response mechanism is designed to be highly selective. For complex architectural violations (fat controllers and high complexity), the system limits its intervention to descriptive repair actions without automatically modifying the code, since simplification heavily relies on specific business contexts. Conversely, for direct database access violations, the transformation pattern is highly structured (moving data access logic from the controller to the service layer). Thus, the system provides an action option that leads to the full execution of auto-refactoring
3) Refactor Engine and Workspace Edit Implementation: When the user selects the auto-refactoring option, the RefactorEngine extracts the code segment containing direct database access and constructs code modifications by moving the data access logic to a more appropriate layer. The system executes them transactionally using the workspace edit mechanism natively provided by the VS Code Extension API, preventing syntax errors during transformation.
4) Code Quality Managerial Features Implementation: Three crucial managerial features were introduced to facilitate broader project management:
   CSmell Summary: Integrated with the VS Code Command API via SummaryManager, it conducts a workspace-wide scan of all PHP files and projects mathematical results into the Output Panel, acting as a technical debt dashboard.
   CSmell Config: Implemented through ConfigManager, it provides a dynamic configuration interface based on native Quick Pick and Input Box features, allowing teams to redefine threshold values. It includes a Reset to Default function to restore standard parameters.
   Ignore Violation: Accommodates intentional deviations through IgnoreManager. When selected, it extracts the violation signature into an internal exclusion list to suppress diagnostic noise without disabling the rule entirely.
5) Testing: The comprehensive results of this operational validation are systematically presented in Table 2.

Table 2. Black-Box Testing Results of The Second Iteration

No	Testing Scenario	Expected Result	Actual Result	Conclusion
1	Violation detected in the editor	System displays Code Action (Quick Fix)	Appeared according to condition	Success
2	User does not select Code Action	No changes occur in the source code	No changes occurred	Success
3	Fat controller violation	System displays recommendation without refactoring	Matched expected condition	Success
4	High complexity violation	System displays recommendation without refactoring	Matched expected condition	Success
5	Direct database access violation	System provides an auto-refactoring option	Detected according to rule	Success
6	Auto-refactoring execution on direct database access	System transforms the code automatically	Changes successfully applied	Success
7	Result after the refactoring process	Code is updated via the workspace edit mechanism	Code successfully updated	Success
8	User executes CSmell Summary	System displays a summary of all project violations	Summary successfully displayed	Success
9	User changes rule threshold via CSmell Config	System saves and applies the new threshold value	Value successfully applied	Success
10	User selects Reset to Default on CSmell Config	System restores the configuration to default values	Configuration successfully restored	Success
11	User selects Ignore This Violation	System saves the violation to the ignored violations list	Violation successfully saved	Success
12	Re-analysis after violation is ignored	The ignored violation is no longer displayed	Violation is not displayed	Success

3.4 User Acceptance Testing (UAT) Analysis

Following technical verification, UAT was conducted involving 10 specialized expert users (active software practitioners with hands-on Laravel and VS Code experience) using a 10-item questionnaire on a 1-to-5 Likert

Journal of Information System Research (JOSH)
Volume 7, No. 4, Juli 2026, pp 1174−1183
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i4.10618

scale. The compiled empirical testing data revealed a highly positive overall average score of 4.18, placing the extension strictly in the "Good" category.

From an architectural evaluation standpoint, the highest score (4.5, Very Good category) was achieved on the statement regarding the system's guidance in logic separation according to Clean Architecture principles, proving its efficacy in mentoring developers to break away from the destructive fat controller anti-pattern. The extension's ability to detect structural violations received a rating of 4.3. Performance-wise, from the end-user perspective, the system demonstrated adequate response speed (4.1) and consistency (4.1) during real-time code modifications, with a non-disruption rating of 4.2. This confirms that the WebAssembly background execution operates seamlessly and unobtrusively, ensuring that the background static analysis does not hinder the IDE's responsiveness during active development. On the user interaction side, Code Actions and restructuring recommendations received a solid score of 4.0 (Good), indicating room for future text-explanation enhancement within the Quick Fix menu.

3.5 Discussion

This section synthesizes the empirical results to highlight the research novelty by directly comparing the findings with previous studies. The Black-Box testing and UAT results confirm that the developed extension successfully transitions code smell mitigation from passive observation to active, in-editor structural manipulation.

When compared to the findings of Wadi et al. [15], who successfully utilized AST for code smell detection in Python, their approach remained limited to general syntax analysis. The results of this research significantly advance their findings by proving that AST can be pushed beyond passive detection to strictly enforce architectural boundaries, specifically Clean Architecture in Laravel. Furthermore, while Nuraminah and Ammar [16] demonstrated the high structural precision of AST for code plagiarism, the empirical results of this study leverage that same precision to accurately isolate complex architectural violations such as filtering out irrelevant global functions during database access detection without yielding false positives.

From a developer tooling perspective, the findings directly address the limitations identified by Alharbi and Alshayeb [22], whose comparative study revealed that most automated refactoring tools still function predominantly as passive diagnostic reporters. The successful implementation of the Code Action Provider and Refactor Engine in this research provides a functional, active alternative that automatically executes transactional logic migration. Finally, while Nugroho et al. [23] validated the importance of applying Clean Architecture to real-world applications, their refactoring process relied heavily on manual execution. The extension developed in this study automates this exact process, ensuring that developers are immediately equipped with a mechanism to extract and migrate infrastructure logic into the correct service layers without breaking existing syntax. Consequently, this extension serves as a proactive architectural mentor rather than a simple error linter.

4. CONCLUSION

This research has successfully designed, developed, and empirically validated a Visual Studio Code Extension that leverages Tree-sitter WebAssembly technology to perform real-time, AST-based static code analysis on Laravel projects. By employing a rule-based approach grounded in Clean Architecture principles, the system effectively mitigates three primary architectural code smells within the presentation layer: fat controllers, high cognitive complexity, and direct database access. The main contributions of this research are divided into theoretical and practical aspects. Theoretically, this study advances the software engineering domain by demonstrating the viability of transitioning from passive syntax observation to active, AST-driven structural manipulation. Practically, by integrating interactive Code Actions, automated code restructuring, and comprehensive managerial features (CSmell Summary and Config), the extension serves not merely as a linter, but as a proactive architectural mentor that actively prevents architectural erosion in real-world environments, as validated by a highly positive UAT score (4.18). However, this study acknowledges certain limitations. The current auto-refactoring capability relies fundamentally on deterministic AST pattern matching. Consequently, a potential failure risk exists when the system encounters highly dynamic Eloquent query builders, variable variables, or deeply abstracted custom facades. In such edge cases, the transformation may be bypassed or require manual developer oversight to prevent unintended logical shifts. Future research should explore integrating lightweight machine learning models to enhance the semantic understanding of dynamic query chains, further minimizing these edge-case risks.

REFERENCES

[1] S. Rochimah, T. R. Hadiningrum, B. D. Mardiana, D. O. Siahaan, J. A. Rizky, and M. S. Ary, “A Maintainability Framework to Ensure the Software Quality in Object-Oriented Programming,” IEEE Access, vol. 13, pp. 195796–195821, 2025, doi: 10.1109/ACCESS.2025.3633265.
[2] R. Li, P. Liang, M. Soliman, and P. Avgeriou, “Understanding software architecture erosion: A systematic mapping study,” Mar. 01, 2022, John Wiley and Sons Ltd. doi: 10.1002/smr.2423.
[3] A. Tornhill and M. Borg, “Code Red: The Business Impact of Code Quality – A Quantitative Study of 39 Proprietary Production Codebases,” 2022. [Online]. Available: https://arxiv.org/abs/2203.04374
[4] “Laravel - The clean stack for Artisans and agents.” Accessed: Apr. 25, 2026. [Online]. Available: https://laravel.com/

Journal of Information System Research (JOSH)
Volume 7, No. 4, Juli 2026, pp 1174−1183
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i4.10618

[5] "PHP: Introduction - Manual." Accessed: Apr. 25, 2026. [Online]. Available: https://www.php.net/manual/en/introduction.php
[6] A. F. Rahmawati and Y. A. Susetyo, "Analisis Quality Code Menggunakan Sonarqube Dalam Suatu Aplikasi Berbasis Laravel," IT-Explore: Jurnal Penerapan Teknologi Informasi dan Komunikasi, vol. 2, no. 2, pp. 99–103, 2023, doi: 10.24246/itexplore.v2i2.2023.pp99-103.
[7] M. Y. Rangkuti, H. R. Ilhamsyah, R. Z. Mutaqin, and M. L. Khoirullah, "Analisis Penerapan Arsitektur Model–View–Controller (MVC) pada Pengembangan Aplikasi Web Berbasis Laravel," Jurnal Teknologi Sistem Informasi, vol. 3, no. 2, pp. 12–21, 2022.
[8] B. Moloudi, "Optimizing Architecture in iOS Development: A Comparative Study of MVVM and VIPER using SwiftUI," Master's thesis, Interactive Technologies, Turku University of Applied Sciences, Turku, Finland, 2025.
[9] A. Rio and F. Brito e Abreu, "PHP code smells in web apps: Evolution, survival and anomalies," J. Syst. Softw., vol. 200, p. 111644, 2023, doi: 10.5281/zenodo.
[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code, 2nd ed. Boston, MA, USA: Addison-Wesley, 2018.
[11] N. S. Filho, "Comparative Analysis between Cognitive Complexity and Cyclomatic Complexity in Software Development," Leaders.Tec.Br, vol. 2, no. 7, doi: 10.5281/zenodo.14879076.
[12] R. Hu, "An Automated Approach to Check Software Architecture Erosion," Master's thesis, Dept. Math. Comput. Sci., Eindhoven University of Technology, Eindhoven, Netherlands, 2023.
[13] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston, MA, USA: Prentice Hall, 2017.
[14] N. S. P. K. Yadati, "Architecture Design (MVVM + Clean Architecture)," Journal of Artificial Intelligence, Machine Learning and Data Science, vol. 1, no. 3, pp. 703–706, Sep. 2023, doi: 10.51219/jaimld/naga-satya-praveen-kumar-yadati/177.
[15] Hamzan Wadi, Kasmawi, and Muhammad Asep Subandri, "Penggunaan Software Metrics Dan Abstract Syntax Tree Untuk Mendeteksi Code Smell Pada Bahasa Pemrograman Python," JEKIN - Jurnal Teknik Informatika, vol. 5, no. 1, pp. 61–69, Jan. 2025, doi: 10.58794/jekin.v5i1.900.
[16] A. Nuraminah and A. Ammar, "Damerau-Levenshtein Distance Algorithm Based on Abstract Syntax Tree to Detect Code Plagiarism," Scientific Journal of Informatics, vol. 11, no. 1, pp. 11–20, Dec. 2023, doi: 10.15294/sji.v11i1.48064.
[17] "Implementing Parser and PSI | IntelliJ Platform Plugin SDK." Accessed: Apr. 11, 2026. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/implementing-parser-and-psi.html
[18]A. Sergeyuk, I. Zakharov, E. Koshchenko, and M. Izadi, "Human-AI Experience in Integrated Development Environments: A Systematic Literature Review," Jan. 2026, doi: 10.1007/s10664-025-10793-0.
[19] "Static Code Analysis: Tools and Best Practices | Cycode." Accessed: Apr. 06, 2026. [Online]. Available: https://cycode.com/blog/static-code-analysis/
[20] "Extension API | Visual Studio Code Extension API." Accessed: Apr. 06, 2026. [Online]. Available: https://code.visualstudio.com/api
[21] S. Kushwah, C. Bansal, S. Rathore, V. Kate, D. Bargal, and D. Vishwakarma, TypeScript: An Open-Source Programming Language with Options for Robust Development and Large-Scale Applications. 2024. doi: 10.1109/ACROSET62108.2024.10743426.
[22] M. Alharbi and M. Alshayeb, "A Comparative Study of Automated Refactoring Tools," IEEE Access, vol. 12, pp. 18764–18781, 2024, doi: 10.1109/ACCESS.2024.3361314.
[23] B. Nugroho, N. F. Azhar, and P. C. Subekti, "Refactoring Aplikasi XYZ Menggunakan Pendekatan Clean Architecture," Information System Journal, vol. 7, no. 01, pp. 20–33, Jun. 2024, doi: 10.24076/infosjournal.2024v7i01.1579.
[24] "Tree-sitter: An incremental parsing system for programming tools." Accessed: Apr. 07, 2026. [Online]. Available: https://tree-sitter.github.io/tree-sitter/
[25] J. J. Wijadi, B. Ghiffar, and I. B. K. Manuaba, "A Study on the Performance of WebAssembly Versus JavaScript in an SVG Editor Environment," in 2024 International Symposium on Parallel Computing and Distributed Systems (PCDS), 2024, pp. 1–5. doi: 10.1109/PCDS61776.2024.10743854.
[26] A. Supriyatna and D. Puspitasari, "Implementation of Extreme Programming Method in Web Based Digital Report Value Information System Design," IJISTECH (International Journal of Information System & Technology), vol. 5, no. 1, pp. 67–75, 2021, doi: 10.30645/ijistech.v5i1.116.
[27] T. Haryati, W. Handini, and Y. M. Aprita, "Penerapan Extreme Programming dalam Pembangunan Sistem Informasi Pembayaran Tagihan PAM pada PAMDES Margakaya Sejahtera Karawang," Jurnal Penelitian Inovatif, vol. 4, no. 1, pp. 129–136, Feb. 2024, doi: 10.54082/jupin.278.
[28] D. J. C. Sihombing, "Enhancing Project Visibility: A Study on Construction Project Dashboard Development with Extreme Programming," Informatika dan Sains, vol. 14, no. 01, pp. 45–53, 2024, doi: 10.54209/infosains.v14i01.3739.
[29] S. Astiti, "RESOLUSI : Rekayasa Teknik Informatika dan Informasi Penerapan Metode Extreme Programming Pada Rancang Bangun Website Company Profile," Media Online, vol. 3, no. 3, pp. 114–124, 2023, [Online]. Available: https://djournals.com/resolusi
[30] I. G. B. Premana Putra, M. Sudarma, and I. B. G. Manuaba, "Penerapan Metode Extreme Programming pada Rancang Bangun Sistem Analisis Sentimen Portal Berita," Jurnal Teknologi Informasi dan Ilmu Komputer, vol. 10, no. 6, pp. 1369–1378, Dec. 2023, doi: 10.25126/jtiik.2023106904.
[31] M. T. Abdillah, I. Kurniastuti, F. A. Susanto, and F. Yudianto, "Implementasi Black Box Testing dan Usability Testing pada Website Sekolah MI Miftahul Ulum Warugunung Surabaya," Jurnal Ilmu Komputer dan Desain Komunikasi Visual, vol. 8, no. 1, pp. 234–242, 2023, doi: 10.55732/jikdiskomvis.v8i1.897.
[32] M. Hennink and B. N. Kaiser, "Sample sizes for saturation in qualitative research: A systematic review of empirical tests," Soc. Sci. Med., vol. 292, p. 114523, 2022, doi: https://doi.org/10.1016/j.socscimed.2021.114523.
[33] S. K. Ahmed, "Sample size for saturation in qualitative research: Debates, definitions, and strategies," Journal of Medicine, Surgery, and Public Health, vol. 5, p. 100171, 2025, doi: https://doi.org/10.1016/j.glmedi.2024.100171.

Journal of Information System Research (JOSH)
Volume 7, No. 4, Juli 2026, pp 1174−1183
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i4.10618

[34] Sugiyono, Metode Penelitian Kuantitatif, Kualitatif, dan R&D. Bandung: Alfabeta, 2013.