ARIT: A Tool for Detecting Code Smells in Clojure Programs

Thiago Alves Laurentino
Federal University of Campina Grande (UFCG)
Campina Grande, Brazil
thiago.laurentino@copin.ufcg.edu.br

Rafael Serey
Federal University of Campina Grande (UFCG)
Campina Grande, Brazil
rafael.serey@splab.ufcg.edu.br

Walber Araújo
Federal University of Campina Grande (UFCG)
Campina Grande, Brazil
walber.araujo@splab.ufcg.edu.br

Natália Rodrigues
Federal University of Campina Grande (UFCG)
Campina Grande, Brazil
natalia.rodrigues@splab.ufcg.edu.br

João Brunet
Federal University of Campina Grande (UFCG)
Campina Grande, Brazil
joao.arthur@computacao.ufcg.edu.br

Abstract

Code smells signal potential quality and maintainability issues in source code. Although research on code smell detection has advanced in object-oriented languages, support for functional languages such as Clojure remains limited despite their growing adoption. Existing Clojure static analysis tools, such as *clj-kondo* and *kibit*, mainly focus on granular language-specific issues, including syntactic inconsistencies, idiomatic refactorings, and localized code patterns, leaving a gap in detecting broader maintainability problems. To address this gap, we present ARIT, a static analysis tool capable of detecting 20 Clojure code smells from a recently proposed catalog validated by practitioners. ARIT supports the early identification of maintainability issues, contributing to software quality engineering practices in functional programming languages.

Demo video: https://zenodo.org/records/20388366

Keywords

Code Smells, Functional Programming, Clojure, Software Quality, Static Analysis

1 Introduction

Assuring software quality is essential for the success and maintainability of modern systems. Various approaches exist to assess software quality, and the identification of code smells has been widely adopted as an effective means of revealing potential design and implementation problems [13]. Code smells indicate suboptimal structures that may hinder readability, evolution, and long-term maintenance [13], and have been extensively studied in the literature, especially in the context of object-oriented (OO) programming [4, 13, 16].

Over the years, a significant body of research has been devoted to the detection and refactoring [19, 24] of code smells in OO systems, resulting in well-established catalogs [3, 13] and a variety of automated tools [8, 17, 23]. These tools typically rely on static analysis techniques to identify structural and design issues, supporting developers in improving software quality. However, this effort has been predominantly focused on OO languages, leaving other paradigms comparatively underexplored.

In contrast, tools for functional programming languages have traditionally emphasized lower-level concerns, such as evaluation strategies, performance, and idiomatic usage [9, 12, 14]. As a result, existing tools such as *clj-kondo* [2] and *kibit* [6] mostly focus on linting and the detection of fine-grained issues, such as syntactic problems or non-idiomatic constructs. While useful, these approaches do not address higher-level design and structural problems that also impact software quality.

Recent studies have shown that functional languages exhibit their own set of code smells, including higher-level issues related to design and architecture. In particular, the seminal work by Veggi et al. [10] identified and categorized code smells specific to functional programming. Building upon this work, our research group recently replicated and extended the proposed methodology in the context of Clojure, resulting in a catalog of Clojure-specific code smells that was systematically constructed and validated [1, 11].

Despite these advances, there is still a lack of tools capable of automatically detecting such higher-level code smells in functional languages. The absence of automated support makes the identification of these issues difficult and impractical, especially in large codebases.

In this context, we present ARIT[1][2], a static analysis tool for detecting code smells in Clojure programs. We developed ARIT based on a catalog-driven approach and, it focuses on identifying higher-level, paradigm-specific issues that are not covered by traditional linters. The tool analyzes source code and reports code smells with precise location information, severity levels, and refactoring suggestions, supporting developers in improving code quality and maintainability.

This document is organized as follows. Section 2 discusses related work on code smell detection and static analysis tools, with emphasis on functional programming languages and the Clojure ecosystem. Next, Section 3 presents ARIT, describing its architecture, analysis pipeline, detection mechanisms, and supported code smells. Section 4 illustrates how the tool can be used in practice through execution examples and configuration options. Section 5 compares ARIT with existing Clojure analysis tools, highlighting its contributions and differences. Finally, Section 6 concludes the paper and outlines directions for future work.

[1] A recursive acronym for ARIT Rapidly Inspects Trees. Additionally, the name serves as a phonetic allusion to arity, a fundamental concept in functional programming.
[2] https://github.com/nufuturo-ufcg/clj-smells-arit

2 Related Work

Code smells have been extensively studied since the work of Martin Fowler [13] introduced the concept in the book *Refactoring: Improving the Design of Existing Code*. In the object-oriented (OO) context, several static analysis and smell detection tools have been proposed, including *PMD*, *SonarQube*, *SpotBugs*, *JDeodorant*, *Designite*, and *JSpIRIT* [8, 17, 20, 23, 24]. These tools combine rule-based analysis, metric extraction, and syntactic inspection to identify maintainability and design problems, such as God Class, Long Method, Primitive Obsession, and Duplicated Code.

In practice, linters, such as *SonarQube* [20], *PMD* [17], *CheckStyle* [5], *Pylint* [18] represent one of the most widely adopted forms of static analysis. These tools rely on predefined rules and lightweight heuristics to identify coding convention violations, syntactic issues, and localized structural problems. While effective within this scope, their analysis is generally limited to low-level constructs and does not encompass higher-order design concerns or system-wide structural issues.

In functional programming languages, however, static analysis tools are generally more focused on idiomatic usage, style consistency, and lightweight defect detection than on higher-level design smells. Existing linters typically report issues such as unused variables, suspicious expressions, reflection warnings, and deviations from language best practices. Tools such as *Credo* [9] for Elixir, HLint [15] for Haskell, and Clippy [22] for Rust, emphasize idiomatic recommendations and code quality assistance rather than structural smell detection.

In the context of Clojure, several tools have been developed to enhance overall code quality through rule-based and static analysis techniques. Widely adopted tools such as *clj-kondo*, *kibit*, *Eastwood*, and *Joker* [2, 6, 12, 14] support developers by detecting potential defects and suspicious constructs, including unused variables, incorrect API usage, reflection-related warnings, and macro expansion issues. Beyond defect detection, these tools also encourage adherence to idiomatic Clojure patterns and community best practices, contributing to greater consistency and maintainability across codebases.

However, similarly to higher-level analyzers in object-oriented ecosystems, there remains a need for tools focused on structural and design-level issues, which are beyond the intended scope of traditional Clojure linters. In this context, ARIT complements existing analyzers by going beyond conventional linting and targeting design and structural problems. It is a static analysis tool for detecting code smells in Clojure programs, based on the catalog of smells identified in our catalog previously mentioned. Its detection rules are derived directly from this catalog, addressing a gap in the current landscape of Clojure static analysis tools, as discussed in the "Comparison with other Tools" section.

3 ARIT Tool

ARIT is a static analysis tool designed to detect code smells in Clojure programs. The tool was developed in Go and uses the goclj library [21] to parse Clojure source code and construct an enriched Abstract Syntax Tree (AST) that enables both syntactic pattern matching and semantic reasoning. ARIT detects Clojure-specific code smells through analysis rules. Each detected smell is reported with precise location information, severity classification (ERROR, WARNING, INFO, HINT), clear descriptions, and actionable refactoring suggestions. The tool supports multiple output formats (text, JSON, HTML, Markdown) and can be configured through YAML files to enable or disable specific rules according to project needs.

3.1 Overview

Figure 1 presents the main ARIT components and their interactions throughout the analysis process. The architecture follows a modular pipeline design that transforms source code into actionable quality insights through the four main components described below.

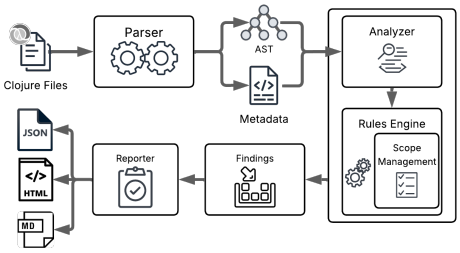


Figure 1: ARIT internal architecture showing the main analysis components and data flow.

Parser. The Parser component is responsible for reading Clojure source files and constructing an enriched AST representation of the code. It leverages the goclj library for robust parsing of Clojure syntax, producing a raw parse tree that captures the syntactic structure of S-expressions. The Parser then enriches this raw tree with additional metadata essential for analysis: node type classification (Symbol, List, Vector, Map, Function Definition, etc.), precise source location information (file, line, column), parent-child relationships for bidirectional traversal, type hints extracted from metadata, and separation of comment nodes from executable code. This enrichment phase transforms the syntactic parse tree into a semantically annotated structure that supports the contextual reasoning required by detection rules.

Analyzer. The Analyzer serves as the orchestrator of the entire detection process. It loads all available detection rules from the Rules Engine and systematically traverses the enriched AST in depth-first order, applying each rule to every node encountered. This straightforward traversal approach leverages the homogeneous nature of Clojure's S-expressions, where both code and data are represented as nested lists. This uniformity eliminates the need for type-specific dispatch methods during AST traversal, as all constructs can be processed through a consistent recursive pattern that matches the recursive nature of S-expressions themselves.

Rules Engine and Scope Management. The Rules Engine implements a pluggable architecture where each catalog smell is operationalized as a distinct rule module. Each rule implements a common interface with a checking method that examines a single AST node and returns zero or more findings if the node matches the smell pattern. Rules combine syntactic pattern matching with semantic reasoning to detect both functional and Clojure-specific

issues. To support this reasoning, rules utilize Scope Management, a collection of lightweight helper functions that query structural properties of the AST without constructing a full scope graph. These utilities include functions for measuring nesting depth, tracking function nesting levels, identifying symbol usage, parsing binding forms (including destructuring), and detecting threading macro contexts. By composing these utilities, rules achieve precise detection while keeping the architecture simple and maintainable.

Reporter. The Reporter transforms the aggregated findings into human-readable or machine-processable output formats. ARIT supports multiple formats tailored to different use cases: plain text for terminal output, Markdown for code review comments and documentation, JSON for integration with automated tools and dashboards, and HTML for visual presentation with embedded code snippets and syntax highlighting. For the HTML format, the Reporter extracts the relevant source code lines surrounding each finding, providing immediate visual context. The Reporter also performs grouping by file, sorting by line number, deduplication to avoid redundant reports, and severity filtering to focus on high-priority issues, ensuring that the output is organized and actionable.

3.2 Running Example

To illustrate how ARIT processes code and detects smells through its internal pipeline, we present a concrete example using the **Immutability Violation** smell from the catalog [11]. This smell occurs when developers introduce direct, unmanaged memory mutations (such as `set!` or Java interop's `aset`), or misuse managed state primitives (such as using `swap!` or `reset!` in local scopes) where pure, immutable alternatives would be more appropriate. Both patterns violate functional programming principles and can introduce subtle concurrency bugs or unexpected side effects.

We chose this smell as our running example because: (i) it is simple to understand even for developers unfamiliar with Clojure, since the concept of immutability violation is universal in functional programming; (ii) it exercises all components of the architecture including the Parser, Analyzer, Scope Management, Rules Engine, and Reporter; (iii) it demonstrates both syntactic pattern matching and semantic reasoning through symbol resolution; and (iv) it represents a paradigm-specific issue that mainstream linters do not detect.

Consider the following Clojure code snippet that exhibits this smell:

Listing 1: Immutability Violation Example

```
(ns example.iv)

(def my-numbers (int-array [10 20 30]))

(defn violate-immutability [arr]
  (aset arr 0 999))

(vec my-numbers)

(violate-immutability my-numbers)

(vec my-numbers)
```

This code defines a namespace `example.iv` with a primitive Java array and a function. The function, `violate-immutability`, uses `aset` to directly mutate the array in memory. Unlike managed concurrency primitives (like `atom`), this represents a severe Immutability Violation because it alters data in-place, causing side effects that leak outside the function's scope and break Clojure's core functional guarantees.

Step 1: Parsing and AST Construction. When ARIT receives this source file, the Parser component begins by invoking the goclj library [21] to tokenize and parse the S-expressions. The library produces a raw parse tree representing the syntactic structure. For the `violate-immutability` function, the raw parse tree includes nodes for the `defn` form, the parameter vector `[arr]`, and the body expression containing the `aset` call.

The Parser then enriches this raw tree with analysis-specific metadata. Each node is augmented with:

- Type classification (e.g., `Symbol`, `List`, `Vector`)
- Source location (line 5 for `violate-immutability`, line 6 for the `aset`)
- Parent references enabling upward traversal
- Extracted metadata such as docstrings or type hints

The enriched AST for the `violate-immutability` function has the following structure:

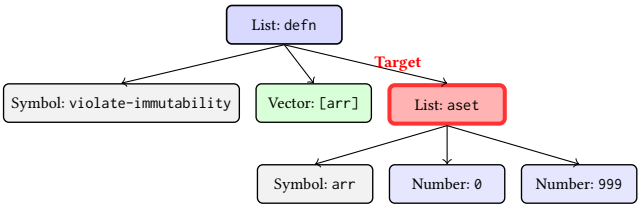


Figure 2: AST structure for the `violate-immutability` function. The `aset` node (highlighted in red) represents the target of detection where a strict immutability linter identifies direct memory mutation on a primitive array.

Each node in this tree includes location metadata pointing to the original source code, enabling precise error reporting. The `aset` node represents the target of detection, where the Immutability Violation rule will identify the problematic pattern.

Step 2: Symbol Resolution via Scope Management. Before detection rules can execute, the Analyzer works with the Scope Management component to build a hierarchical scope structure that mirrors Clojure's lexical scoping. As the Analyzer traverses the AST, it maintains a scope stack:

- **Namespace scope**: Contains `my-numbers` (defined by `def`)
- **Function scope for violate-immutability**: Contains parameter `arr`

When the Immutability Violation rule examines the `aset` node, it needs to determine the origin of the `arr` reference. The rule invokes the Scope Management lookup mechanism, which:

(1) Searches the current (function) scope → finds `arr` as a parameter
(2) Returns metadata:

```
{"name": "arr", "type": "parameter", "
    scope": "violate-immutability"}
```

This lookup confirms that `arr` is passed as an argument. Mutating parameters directly is particularly problematic in functional programming, which strengthens the smell detection: the function is not pure and will silently alter data structures owned by the caller.

Step 3: Rule Application and Detection. With the enriched AST and scope context available, the Analyzer invokes the Immutability Violation rule. The rule implementation leverages predefined sets of mutating symbols and contextual data provided by the Scope Management component:

Listing 2: Immutability Violation Rule Implementation Snippet

```
var mutatingSymbols = map[string]struct{}{
    "set!":           {},
    "alter-var-root": {},
    "aset":           {},
    // ... other symbols omitted for brevity
}

func (r *ImmutabilityViolationRule) Check(node *reader
    .RichNode, context map[string]interface{},
    filepath string) *Finding {
    if node.Type == reader.NodeList && len(node.
        Children) > 0 {
        symbol := node.Children[0]
        if symbol.Type == reader.NodeSymbol {

            // Check for direct memory/state mutation
            if _, isMutating := mutatingSymbols[symbol.
                Value]; isMutating {
                return &Finding{
                    RuleID:   r.ID,
                    Message:  fmt.Sprintf("Found state
                        mutation function call: `%s`.
                        This can lead to side effects
                        and violates immutability
                        principles.", symbol.Value),
                    Filepath: filepath,
                    Location: symbol.Location,
                    Severity: r.Severity,
                }
            }

            // Contextual checks (e.g., 'def' inside
                local scope)
            // are evaluated here using the 'context'
                map...
        }
    }
    return nil
}
```

The rule applies this logic to the `aset` node at line 6:

(1) Confirms the node is a list and extracts the first child (the function symbol)
(2) Checks if the symbol exists in the `mutatingSymbols` map, which tracks known direct mutations like `aset`
(3) Detects the `aset` usage and immediately creates a finding populated with location metadata and severity levels

Step 4: Report Generation. After the Analyzer completes its traversal and collects all findings, it passes them to the Reporter component. For our example, the Reporter receives one finding for the Immutability Violation at line 6.

The Reporter performs several transformations:

- Groups findings by file (`example/iv.clj`)
- Sorts them by line number
- Extracts surrounding source code for context
- Formats output according to the selected format

For the plain text format, the output appears as shown in Listing 3:

Listing 3: Plain Text Output Example

```
File: example/iv.clj

Line 6: Immutability Violation (WARNING)
  (aset arr 0 999))
   ^^^^
Message: Found state mutation function call: `aset`.
    This can lead to side effects and violates
    immutability principles.
```

For the JSON format (useful for tool integration), the output includes structured data, as demonstrated in Listing 4:

Listing 4: JSON Output Example

```
{
  "findings": [
    {
      "ruleId": "immutability-violation",
      "filepath": "example/iv.clj",
      "location": {
        "line": 6,
        "column": 4
      },
      "severity": "Warning",
      "message": "Found state mutation function call: `
          aset`. This can lead to side effects and
          violates immutability principles."
    }
  ]
}
```

Finally, for the HTML format, the output includes structured data and code snippets of detected code smells, facilitating human analysis, as illustrated in Figure 3.

This example demonstrates the complete flow through ARIT's architecture: from source code to enriched AST with scope resolution, through rule-based pattern detection using context-aware implementations, down to actionable formatted output. The modular design ensures that each component performs a well-defined role while maintaining clear interfaces, enabling both accurate detection and straightforward extensibility of the ruleset.

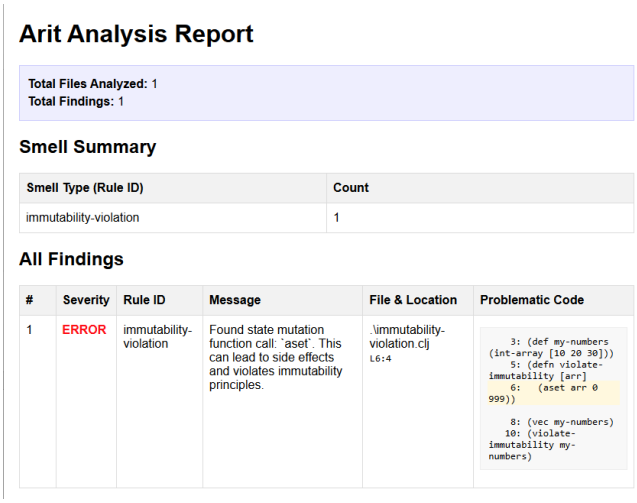
3.3 Currently Detected Code Smells

Currently, ARIT detects 20 of 34 smells, as shown in Table 1.

The Clojure-specific smells address issues that arise from misuse of language features and idiomatic patterns unique to Clojure. For example, *Immutability Violation* identifies cases where developers use mutable operations (`swap!`, `reset!`, `alter`) in contexts where

Table 1: Clojure-specific code smells and their detectability by ARIT (✓ = Detectable, × = Not Detectable)

Unnecessary Macros (×)	Immutability Violation (✓)	Improper Emptiness Check (✓)
Map With Nil Values (×)	Unnecessary into (✓)	Conditional Build-Up (✓)
Verbose Checks (✓)	Production doall (✓)	Redundant do Block (✓)
Thread Ignorance (✓)	Nested Forms (✓)	Direct Usage of clojure.lang.RT (✓)
Non-Idiomatic Record Construction (×)	Misuse of Dynamic Scope (×)	Implicit Namespace Dependencies (✓)
Namespace Load Side Effects (✓)	Blocking Inside Go (✓)	Nested Atoms (×)
Single-segment Namespace (✓)	Dynamic Scoped Singleton Resource (×)	Overengineering with core.async (×)
Excessive Refers (✓)	Unnecessary Laziness (×)	Relying on Load-Time Side Effects (×)
Monolithic Namespace Split (✓)	Unmanaged Resource I/O (✓)	Refs in Dependency Vector (×)
Misuse of Channel Closing Semantics (✓)	Misused Threading (×)	Marker Protocol (×)
Multiple Evaluation in Macros (✓)	Case with Non-Literal Test Values (×)	Non-Idiomatic Parameter Binding (×)
Private Multimethods (✓)		



Arit Analysis Report

Total Files Analyzed: 1
Total Findings: 1

Smell Summary

Smell Type (Rule ID)	Count
immutability-violation	1

All Findings

#	Severity	Rule ID	Message	File & Location	Problematic Code
1	ERROR	immutability-violation	Found state mutation function call: `aset`. This can lead to side effects and violates immutability principles.	.\immutability-violation.clj L6:4	3: (def my-numbers (int-array [10 20 30])) 5: (defn violate-immutability [arr] 6: (aset arr 0 999)) 8: (vec my-numbers) 10: (violate-immutability my-numbers)

Figure 3: Example of the HTML report highlighting detected code smells.

immutable alternatives would be more appropriate, and *Thread Ignorance* detects uses of threading macros (`->`, `->>`) where the threaded value is never used. *Production doall* warns about unnecessary forcing of lazy sequences in production code that could benefit from lazy evaluation's performance advantages.

4 Example of Use

This section describes how ARIT is used in practice. We show the main execution flow, the available report formats, and how developers can customize the analysis through the command-line interface and a YAML configuration file.

ARIT is delivered as a command-line tool that analyzes Clojure source code and reports code smells with precise location data. To use ARIT, the user first downloads the repository and builds the binary with Go:

```
git clone \
  https://github.com/nufuturo-ufcg/clj-smells-arit
cd clj-smells-arit
go build -o arit .
```

Once the binary is available, the user can analyze a single file, multiple files, or a whole directory recursively. For example, executing:

```
./arit src/
```

runs the analysis on all Clojure files under `src/`. The default output is a human-readable text summary.

ARIT also supports several report formats. The following command generates a JSON report suitable for CI/CD integration:

```
./arit --format json src/ > analysis-results.json
```

For a richer review, the user can produce an HTML report:

```
./arit --format html src/ > report.html
```

ARIT provides auxiliary commands to inspect and configure the analysis. The command:

```
./arit list-rules
```

prints all available analysis rules, while:

```
./arit genconf
```

generates a default `.arit.yaml` configuration file that can be edited to enable or disable specific rules and tune rule parameters.

A typical ARIT report includes the rule identifier, a brief description of the detected smell, file path, start line and column, and severity level (ERROR, WARNING, INFO, HINT).

For example, ARIT may output a warning such as:

```
[WARNING] blocking-inside-go: Blocking function detected
within the GO block go. [src/core.clj:42:8]
```

or a hint such as:

```
[HINT] thread-ignorance: Detects nested function calls
that would benefit from threading macros (-> or ->>).
[src/core.clj:45:8]
```

and an error for immutability violation such as:

```
[ERROR] immutability-violation: Found state mutation
function call: `set!`. This can lead to side effects
and violates immutability principles. [src/core.clj:50:4]
```

Designed to integrate seamlessly with the existing Clojure ecosystem, this versatile usage model allows ARIT to function as a standalone command-line analyzer, an automated checker in CI/CD pipelines, or an integrated component within modern IDEs via clojure-lsp [7]. In all scenarios, its structured JSON or HTML outputs can be readily consumed by external tools.

5 Comparison with Other Tools

The current version of the tool is fully functional and has already been executed on real projects, where it was able to identify multiple occurrences of the targeted smells. For example, in our preliminary analyses, ARIT successfully flagged complex instances of *Immutability Violation* and *Thread Ignorance* in existing open-source Clojure repositories, highlighting structural problems that often go unnoticed by conventional linters. Although further large-scale evaluations are still required, these initial observations suggest that the approach is promising both as a research contribution and as a practical aid for developers.

To better understand ARIT's practical contribution, it is essential to contextualize it within the existing ecosystem of Clojure quality tools, most notably *clj-kondo* [2], *kibit* [6], and *Eastwood* [12]. While these tools are widely adopted and highly effective for maintaining code health, they primarily focus on lower-level concerns, leaving a gap for the higher-level design analysis that ARIT aims to fill.

clj-kondo [2] is widely adopted in practice as a fast, first line of defense, providing instantaneous feedback directly in the developer's editor. While highly effective at catching syntax errors, arity mismatches, and unused declarations during everyday coding, its analysis is confined to low-level correctness. In real-world projects, it does not evaluate higher-level functional design decisions, such as state mutation boundaries or structural complexity.

kibit [6] serves as a practical aid for developers seeking to write more idiomatic Clojure code. By suggesting alternative expressions (e.g., using `when` instead of an `if` without an `else` branch), it helps maintain consistent coding styles across a codebase. However, its scope is strictly limited to localized, syntactic micro-optimizations, missing broader architectural or structural code smells.

Eastwood [12] is frequently used in CI/CD pipelines to uncover subtle execution defects, such as reflection warnings or incorrect type hints. Because it deeply analyzes code execution anomalies, it achieves high accuracy. Yet, similar to clj-kondo, its practical contribution lies in preventing runtime defects rather than identifying design degradation and maintainability issues in large codebases.

In contrast to these tools, **ARIT** is specifically designed to detect higher-level, paradigm-specific code smells based on a systematically validated catalog. As evidenced by its execution on real projects, ARIT identifies architectural and maintainability issues such as *Immutability Violations*, *Misuse of Dynamic Scope*, *Thread Ignorance*, and *Production doall Usage* that directly impact the long-term maintainability of functional systems. By constructing an enriched AST with scope analysis to perform deep pattern matching, ARIT avoids the overhead of full code evaluation while focusing on design degradation rather than simple unused variables or execution defects. Furthermore, ARIT's customizable YAML configuration and multiple machine-readable output formats (JSON, HTML, Markdown) make it uniquely suited for enforcing functional design standards in automated CI/CD pipelines, effectively complementing the existing ecosystem of Clojure tools.

6 Conclusion

In this paper we introduce ARIT, a static analysis tool for detecting code smells in Clojure programs. Driven by a systematically validated catalog of functional code smells, ARIT goes beyond traditional linting by analyzing source code to uncover paradigm-specific design flaws, such as *Immutability Violations* and *Misuse of Dynamic Scope*. Through its modular architecture and support for multiple output formats, ARIT provides actionable feedback that assists developers in improving the structural quality of their systems, both during interactive code reviews and in automated pipelines.

As future work, we plan to expand ARIT's capabilities. First, we will focus on implementing the remaining rules from the catalog. Second, we intend to develop a high-level API for rule creation, lowering the barrier for developers to define custom, project-specific rules without requiring deep knowledge of the tool's internal architecture. Third, we aim to implement support for the SARIF (Static Analysis Results Interchange Format) standard, enabling seamless integration with a broader ecosystem of CI/CD pipelines, code quality platforms, and security dashboards. Finally, we plan to introduce automated refactoring suggestions and dedicated IDE plugins to bring ARIT's architectural feedback directly into the developer's everyday workflow.

Artifact Availability

The ARIT source code is licensed under the MIT license and is available on GitHub at https://github.com/thlaurentino/arit. The ARIT tool, including its source code, executables, and documentation, is available at: https://doi.org/10.5281/zenodo.22149097. The demo video is available at: https://zenodo.org/records/20388366.

Acknowledgements

This work was developed under the NuFuturo project, a collaboration between the Federal University of Campina Grande and Nubank.

References

[1] Walber Araújo, José Truta, Lucas Vegi, Marco Tulio Valente, and João Brunet. 2026. Code Smells in Clojure: Initial Findings from a Grey Literature Review. arXiv:2605.24049 [cs.SE] https://arxiv.org/abs/2605.24049

[2] Michiel Borkent. 2019. clj-kondo: A linter for Clojure code that sparks joy. https://github.com/clj-kondo/clj-kondo. Accessed: 2026-04-02.

[3] William J. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons.

[4] Alexander Chatzigeorgiou and Anastasios Manakos. 2014. Investigating the evolution of code smells in object-oriented systems. *Innovations in Systems and Software Engineering* 10, 1 (2014). doi:10.1007/s11334-013-0205-z

[5] Checkstyle Team. 2001. Checkstyle: A tool for checking Java source code. https://checkstyle.sourceforge.io/. Accessed: 2026-04-02.

[6] clj commons. 2012. Kibit: Static code analyzer for Clojure, suggesting idiomatic code improvements. https://github.com/clj-commons/kibit. Accessed: 2026-04-02.

[7] clojure-lsp Contributors. 2019. clojure-lsp: A Language Server for Clojure(script). https://github.com/clojure-lsp/clojure-lsp. Accessed: 2026-04-12.

[8] The SpotBugs Community. 2016. SpotBugs: Static Analysis Tool for Java. https://spotbugs.github.io/. Accessed: 2026-04-04.

[9] Credo Contributors. 2014. Credo: A static code analysis tool for the Elixir language. https://github.com/rrrene/credo. Accessed: 2025-05-28.

[10] Lucas Francisco da Matta Vegi and Marco Tulio Valente. 2023. Understanding code smells in Elixir functional language. *Empirical Software Engineering* 28, 4 (2023). doi:10.1007/s10664-023-10343-6

[11] Walber Wesley Félix de Araújo Filho, José do Bomfim Truta Neto, Rafael Antônio de Lucena Serey, Thiago Alves Laurentino, and João Arthur Brunet Monteiro. 2025. clj-smells-catalog: Catalog of Clojure-related code smells. https://github.com/nufuturo-ufcg/clj-smells-catalog. NuFuturo - UFCG. Accessed: 2026-04-01.

[12] Jonas Enlund. 2014. Eastwood: A Clojure lint tool. https://github.com/jonase/eastwood. Accessed: 2026-04-02.

[13] Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

[14] Cyril G. 2016. Joker: A small Clojure interpreter and linter. https://github.com/candid82/joker. Accessed: 2026-04-02.

[15] Neil Mitchell. 2006. HLint. https://github.com/ndmitchell/hlint. Haskell source code suggestions Accessed: 2026-04-15.

[16] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5, 1 (2017). doi:10.1186/s40411-017-0041-1

[17] PMD Project. 2005. PMD: An extensible static code analyzer. https://pmd.github.io/. Accessed: 2026-04-02.

[18] Pylint Contributors. 2001. Pylint: A Static Code Analysis Tool for Python. https://pylint.readthedocs.io/. Accessed: 2026-04-04.

[19] Daniel Ramos, Railana Santana, and Ivan Machado. 2025. DANTES: Automating Detection and Refactoring of Test Smells. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil. doi:10.5753/sbes.2025.11611

[20] SonarSource. 2008. SonarQube: Continuous inspection of code quality. https://www.sonarsource.com/products/sonarqube/. Accessed: 2026-04-02.

[21] Caleb Spare. 2014. goclj: A Clojure parser and analyzer written in Go. https://github.com/cespare/goclj. Accessed: 2026-03-25.

[22] The Rust Project Developers. 2015. Clippy: A Collection of Lints to Catch Common Mistakes and Improve Your Rust Code. https://github.com/rust-lang/rust-clippy. Official Rust linter, Accessed: 2026-04-15.

[23] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. JDeodorant: Identification and Removal of Type-Checking Bad Smells. https://github.com/tsantalis/JDeodorant. Accessed: 2026-04-02.

[24] Santiago Vidal, Hernan Vazquez, J. Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. 2015. JSpIRIT: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*. 1–6. doi:10.1109/SCCC.2015.7416572 Accessed: 2026-04-04.