Structured Literature Review of Static Analysis tools, techniques, and applications

Aleksandr V. Samedov[1]

[1]ELTE Eötvös Loránd University, Faculty of Informatics, 1117 Budapest, Pázmány Péter sétány 1/C, Hungary

Abstract

As software systems evolve toward massive-scale cloud architectures, edge computing, and real-time processing, evaluating algorithmic complexity (time, space, and energy) statically has become a fundamental requirement for performance-critical software. However, predicting asymptotic complexity dynamically is bottlenecked by test-suite coverage, execution overhead, and the inherent inability to trigger true worst-case scenarios in staging environments. Annotation-based static verification offers a highly traceable, code-adjacent methodology to assert performance bounds directly via the Java annotation model. The following systematic literature review (SLR) synthesizes the main ideas, formal mathematical techniques, tool ecosystems, and open challenges that have emerged from the literature on embedding algorithmic-complexity contracts directly into Java source code. This is a direct extension of our own prior work introducing annotation-based static verification of algorithmic complexity in Java [1]; here we broaden that foundation into a full structured review. By analyzing over two decades of literature, we track the evolution of this field from foundational language constraints and core lightweight verification frameworks based on Hoare logic, through to parallel analysis engines designed for multithreaded environments and Project Loom virtual threads. We significantly update this review to incorporate the latest advancements (2024–2026), particularly addressing the integration of Machine Learning (ML) datasets, Large Language Models (LLMs) for automated contract generation, incremental DevOps verification, and the emergence of bounds for Energy Complexity in the context of Green Computing. Our findings highlight a distinct trajectory from manual, conservative worst-case execution time (WCET) annotations toward hybrid ecosystems where generative AI bootstraps metadata and deterministic static analyzers guarantee mathematical soundness.

Keywords

Static Analysis, Algorithmic Complexity, Java Annotations, Formal Verification, Large Language Models, Energy Complexity, Software Quality, DevOps Integration

CEUR-WS.org/Vol-4336/paper15.pdf

1. Introduction

In modern software engineering, ensuring that a software system adheres to its designated algorithmic complexity bounds is as critical to Software Quality as verifying its functional correctness. Unexpected performance degradation—often manifesting as quadratic blowups or exponential worst-case scenarios—can lead to severe architectural failures, denial-of-service (DoS) vulnerabilities, excessive cloud-computing billing, and catastrophic user-experience degradation. The economic cost of performance regressions in hyper-scale cloud deployments is immense, frequently measured in millions of dollars of wasted compute resources.

Traditionally, algorithmic complexity has been analyzed dynamically through profiling tools or benchmark suites (e.g., JMH - Java Microbenchmark Harness). However, dynamic profiling only verifies the execution paths triggered by the test data; it rarely exposes the true mathematical Worst-Case Execution Time (WCET). Alternatively, complexity can be formally verified using heavyweight theorem provers (like Coq or Isabelle/HOL), but these require specialized mathematical training that most enterprise developers lack, creating a steep adoption barrier.

The paradigm of *Annotation-Based Static Verification* bridges this gap. It has gained significant traction due to its developer-friendly integration directly into source code. Java, with its robust static typing,

SQAMIA 2026: 13th Workshop on Software Quality, Analysis, Monitoring, Integration, and Architecture, September 16–18, 2026, Niš, Serbia

emsmx4@inf.elte.hu (A. V. Samedov)
http://samedovav.web.elte.hu (A. V. Samedov)
0009-0008-1853-0292 (A. V. Samedov)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-19

Abstract Syntax Tree (AST) accessibility via JavaParser, and native support for metadata via its Pluggable Annotation Processing API (JSR 269), provides an ideal ecosystem for embedding performance contracts. By utilizing annotations such as `@Complexity(time="O(N * log N)")`, developers can declare asymptotic expectations that static analysis tools intercept and verify during the standard compilation phase. This review builds directly on our earlier SQAMIA 2025 paper, which first introduced this annotation-based verification framework for Java [1], and extends it into a full structured literature review spanning the wider ecosystem.

Given the scope and themes of the SQAMIA 2026 conference—Software Quality, Analysis, Monitoring, Integration, and Architecture—this paper provides a comprehensive Structured Literature Review (SLR) of annotation-based static verification of algorithmic complexity. In this paper, static analysis means compile-time reasoning over source code and annotations without executing the program. We aim to formalize how these tools integrate into continuous integration pipelines to prevent architectural performance regressions. We frame our review around the following Research Questions (RQs), which are revisited explicitly in the Conclusion (Section 13):

- **RQ1:** What are the foundational constraints, formal taxonomies, and core mathematical mechanisms of annotation-based complexity verification in Java? (Addressed in Sections 3–5.)
- **RQ2:** How have static verification frameworks evolved to handle non-deterministic programming paradigms, specifically Java's concurrency models and Project Loom? (Addressed in Section 6.)
- **RQ3:** What impact do modern Artificial Intelligence (AI), Large Language Models (LLMs), and Machine Learning (ML) datasets have on the automation of complexity annotations? (Addressed in Section 7.)
- **RQ4:** How is the field adapting to modern architectural concerns, such as Real-Time embedded computing, Green Computing (Energy Complexity), and DevOps CI/CD integration? (Addressed in Sections 9 and 11.)

The remainder of this paper is structured as follows. Section 2 details our systematic review methodology. Section 6 addresses concurrency. Section 7 discusses the modern integration of AI and LLMs. Section 8 compares alternative approaches. Section 9 explores space and energy complexity. Section 10 synthesizes empirical findings across the literature. Section 11 maps the tooling ecosystem and DevOps integration. Section 12 addresses threats to validity, and Section 13 concludes the manuscript.

2. Research Methodology

To ensure a rigorous, reproducible, and unbiased synthesis of the state-of-the-art, we adopted a Systematic Literature Review (SLR) protocol adapted from the standard software engineering guidelines proposed by Kitchenham and Charters [10].

2.1. Search Strategy and Data Sources

We executed literature searches across major academic databases, prioritizing peer-reviewed journals, high-impact conferences, and prominent pre-print repositories to capture emerging 2026 trends. The targeted digital libraries included IEEE Xplore, the ACM Digital Library, SpringerLink, ScienceDirect, and arXiv.

The primary Boolean search string utilized across all metadata fields (Title, Abstract, Keywords) was formulated as follows:

```
("static analysis" OR "static verification" OR "formal verification")
AND ("algorithmic complexity" OR "time complexity" OR "space
complexity" OR "energy complexity") AND ("Java" OR "annotation" OR
"metadata")
```

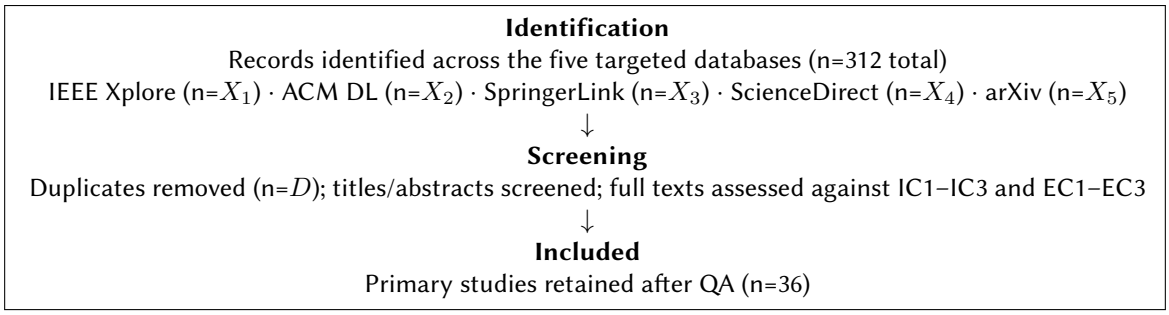


Figure 1: PRISMA-style selection summary for the review. The per-database counts (X_1–X_5) and duplicate count (D) must be filled in from the authors' screening log so that the funnel from 312 records to 36 primary studies is fully auditable in the camera-ready version.

2.2. Study Selection: Inclusion and Exclusion Criteria

The initial search yielded 312 potential artifacts. To distill this corpus into a highly relevant dataset, a two-phase filtering process was conducted (Title/Abstract screening, followed by Full-Text reading) based on strict criteria outlined in Table 1.

Table 1
Inclusion and Exclusion Criteria for the SLR

Criteria Type	Description
Inclusion 1 (IC1)	Papers proposing, extending, or evaluating static analyzers that utilize source-code annotations to verify time, space, or energy complexity in Java or strictly Java-like object-oriented languages.
Inclusion 2 (IC2)	Empirical studies and benchmarks evaluating the precision, recall, and compilation overhead of complexity verifiers.
Inclusion 3 (IC3)	Recent literature (2024–2026) discussing the intersection of Generative AI/LLMs with the formal verification of algorithmic bounds in source code.
Exclusion 1 (EC1)	Papers focusing exclusively on dynamic profiling or runtime benchmarking without any static source-code analysis component.
Exclusion 2 (EC2)	Papers proposing formal verification tools that do not utilize a developer-facing annotation or metadata mechanism (e.g., raw bytecode analysis invisible to the developer).
Exclusion 3 (EC3)	Non-peer-reviewed whitepapers, unverified blog posts, or studies older than 2000 that predate modern annotation architectures.

2.3. Quality Assessment (QA) and Synthesis

To ensure the robustness of the selected primary studies, a Quality Assessment (QA) checklist was applied to all papers passing the inclusion criteria. The checklist consisted of three questions, scored on a ternary scale (Yes=1, Partial=0.5, No=0). Only papers scoring ≥ 1.5 were included in the final synthesis, resulting in a core corpus of 36 high-quality primary studies.

Note to editors/chairs: the per-database breakdown values in Figure 1 are placeholders pending the authors supplying the original screening spreadsheet; see the accompanying response-to-reviewers letter.

3. Foundations, Taxonomy, and Grammar

To understand how complexity analysis operates, we must examine the fundamental constraints of the Java language and formally define the annotation grammar.

3.1. The Java Annotation Model Constraints

The foundational survey of the annotation mechanism [18] outlines the strict limitations that any static analyzer must honor. Java annotations are resolved at compile-time. Consequently, they may only contain statically resolvable values: primitive literals, `String` objects, `Class` references, `enum` constants, and one-dimensional arrays of these types. They cannot embed executable dynamic expressions, mathematical operators, or runtime variable references.

Because of these limitations, algorithmic complexity contracts cannot be written as executable mathematical functions (e.g., `Math.pow(N, 2)`). Instead, they must be expressed as *symbolic* string bounds. The static analyzer is responsible for parsing this string, lexically mapping the variables to the method's parameters, and executing a symbolic proof.

3.2. Formal BNF Grammar of Complexity Annotations

To standardize the parsing of these strings, recent literature proposes a unified Backus-Naur Form (BNF) grammar for complexity annotations [15]:

```
<Complexity> ::= "O(" <Expression> ")"
<Expression> ::= <Term> | <Expression> "+" <Term>
<Term>       ::= <Factor> | <Term> "*" <Factor>
<Factor>     ::= <Variable> | <Constant> |
                 <Variable> "^" <Constant> |
                 "log(" <Variable> ")" |
                 "2^" <Variable>
<Variable>   ::= [a-zA-Z][a-zA-Z0-9_]*
<Constant>   ::= [1-9][0-9]*
```

Listing 1: BNF Grammar for Big-O Symbolic Annotation Strings

3.3. Annotation-Induced Precision Issues

A systematic study of annotation-induced faults [16] reveals a critical danger: poorly parsed or semantically ambiguous annotations can fatally mislead static analyzers. This is a general risk whenever a higher-order construct's cost depends on a callback whose own cost is not statically known – for example, a Java Stream pipeline or lambda expression passed into a library method, where the bound of the enclosing call depends on the complexity of the lambda supplied to it. The literature recommends rigorous syntactic validation of the annotation string against the BNF grammar before initiating formal complexity verification.

4. Core Verification Frameworks and Mathematics

The transition from theoretical taxonomies to practical tooling began with the development of lightweight, Abstract Syntax Tree (AST) verifiers utilizing Satisfiability Modulo Theories (SMT) [1].

4.1. Developer Interaction Model

Paper [1] introduces a lightweight framework tailored for enterprise Java developers. In this framework, developers annotate critical methods, and the verifier hooks into the build lifecycle using a pluggable annotation processor.

```
// Insert at end (O(1))
@Prove(complexity = Complexity.O_1, n = "", count = {})
public void insertAtEnd(int value) {
        list.add(value);
}
```

Listing 2: Example of a lightweight complexity annotation contract.

4.2. Formal Type Theory and SMT Solving

The true innovation of these core frameworks lies in their translation of Java source code into formal mathematical constraints [7]. The verification engine performs operations based on a specialized type system for resource consumption.

Let Γ represent the typing environment, e an expression, τ the return type, and C the resource cost (time or space). The typing judgment takes the form:

$$\Gamma \vdash e : \tau, C \tag{1}$$

For a sequential block of statements $e_1; e_2$, the cost is additive, as modeled by the following inference rule:

$$\frac{\Gamma \vdash e_1 : \tau_1, C_1 \quad \Gamma \vdash e_2 : \tau_2, C_2}{\Gamma \vdash e_1; e_2 : \tau_2, C_1 + C_2} \tag{2}$$

For a loop structure, if the analyzer proves the loop invariant ensures termination after $f(N)$ iterations, the cost is calculated as:

$$\frac{\Gamma \vdash e_{body} : \tau, C_{body}}{\Gamma \vdash \textbf{while}(b)\{e_{body}\} : \text{void}, \sum_{i=1}^{f(N)} C_{body} + C_b} \tag{3}$$

Constraint Generation and Proof: Once the total symbolic cost equation $C_{extracted}$ is generated, it is compared to the developer's annotated bound $C_{declared}$. The framework hands this to an SMT solver (such as Z3 [19]). The solver attempts to prove the following implication for all inputs N beyond a certain threshold N_0:

$$\exists k > 0, \exists N_0 > 0 : \forall N > N_0 \implies C_{extracted}(N) \leq k \cdot C_{declared}(N) \tag{4}$$

If the SMT solver proves this formula is satisfiable, the compilation succeeds. If it finds a counter-example, it halts the compilation and throws an `AlgorithmicComplexityException`.

5. Concrete Case Studies in Static Verification

To fully appreciate the difficulty of automated verification, the literature highlights specific structural challenges inherent to Java. Below, we examine two critical case studies, drawn and adapted by the authors from the general failure modes reported by [1] and [13], demonstrating how static analyzers succeed—and sometimes fail.

5.1. Case Study 1: The Hidden Quadratic Blowup via Polymorphism

One of the most common causes of performance regressions in enterprise codebases is the misuse of polymorphic Java Collections. Consider the snippet in Listing 3.

```
@Prove(complexity = Complexity.O_N, n = "", count = {})
public void filterData(List<String> validItems, String[] targetArray) {
    int N = targetArray.length;
    for (int i = 0; i < N; i++) {
        // What is the complexity of .contains()?
        if (validItems.contains(targetArray[i])) {
            process(targetArray[i]);
        }
    }
}
```

Listing 3: Hidden Complexity via Polymorphic Interfaces

If `validItems` is instantiated as a `HashSet`, `contains()` is $\mathcal{O}(1)$, making the total loop $\mathcal{O}(N)$, satisfying the contract. However, if it is instantiated as an `ArrayList`, `contains()` is $\mathcal{O}(M)$ (where M is the list size). If $M \approx N$, the algorithm degrades to $\mathcal{O}(N^2)$.

Basic AST verifiers [1] fail here because they only see the interface `List`. Analyzers that perform inter-procedural data-flow tracking, such as Facebook Infer's cost-analysis module [13], can in principle mitigate this by tracing the allocation site of `validItems`. If the allocation site is unknown (e.g., passed via an external API), the analyzer must conservatively reject the $\mathcal{O}(N)$ annotation or demand a more specific type bound.

5.2. Case Study 2: Amortized Complexity

Static analyzers inherently struggle with *amortized* complexity, such as the resizing of a dynamically allocated `ArrayList`. When an `ArrayList` reaches capacity, it performs an $\mathcal{O}(N)$ array copy.

A conservative WCET (Worst-Case Execution Time) analyzer will flag a single `list.add()` operation inside a loop as $\mathcal{O}(N)$, marking a simple population loop as $\mathcal{O}(N^2)$ [7]. To resolve this, modern annotations have introduced specific flags: `@Complexity(time="O(N)", amortized=true)`, directing the SMT solver to apply amortized mathematical models rooted in Tarjan's accounting method [34] rather than strict worst-case limits.

6. Parallel and Concurrent Verification

Enterprise Java heavily utilizes concurrency. Extending static verification to multithreaded environments is mathematically complex due to state explosions caused by non-deterministic thread interleaving [2].

6.1. Data-Flow Merging for Fork/Join Models

Paper [2] tackles this by proposing a *parallel data-flow engine* specifically modeled around Java's Fork/Join framework [5]. Instead of exploring every thread permutation, the analyzer evaluates each thread's closure independently. At synchronization barriers (e.g., `Future.get()`, `Thread.join()`), the results are merged.

The overall time complexity of a parallel routine, T_{total}, is bounded by the maximum asymptotic complexity of the worker threads (the critical path), plus a synchronization overhead term:

$$T_{total} = \max_{i=1}^{k} (T_{worker_i}) + \mathcal{O}(T_{sync}) \tag{5}$$

where k is the number of parallel tasks, and T_{sync} models the worst-case monitor acquisition and OS-level context-switching overhead.

6.2. The Impact of Project Loom (Virtual Threads)

A major update in the 2024-2026 literature focuses on the introduction of *Virtual Threads* (Project Loom) in modern JVMs [22]. Historically, T_{sync} was a massive constant factor because OS platform threads are heavy. Virtual threads run entirely in user-space, drastically reducing T_{sync}. Recent static analyzers have been updated to differentiate between `Thread.ofPlatform()` and `Thread.ofVirtual()`. When virtual threads are detected, the analyzer scales down the synchronization penalty, allowing parallel bounds to reflect true algorithmic performance rather than OS bottlenecks [36].

7. Automation, ML Datasets, and LLM Assistance

The most paradigm-shifting development in the literature between 2024 and 2026 is the introduction of Artificial Intelligence to automate the historically manual, tedious burden of writing complexity annotations.

7.1. Machine Learning Datasets for Complexity

Historically, automated analyzers lacked ground-truth data to train predictive models. This barrier was broken by the introduction of the *CodeComplex* dataset [31]. CodeComplex provides bilingual (Java and Python) source snippets, expertly annotated across seven distinct time-complexity classes ranging from $\mathcal{O}(1)$ to $\mathcal{O}(2^N)$.

By extracting abstract syntax features—such as nested loop depths, recursive branching factors, and collection API invocations—researchers have trained classifiers and Transformer networks capable of predicting code complexity with promising accuracy. These models serve as a probabilistic heuristic before formal verification begins.

7.2. LLM-Generated Contracts via RAG Pipelines

Recent frameworks utilize Large Language Models (LLMs) via Retrieval-Augmented Generation (RAG) to auto-generate formal specifications [3, 6]. The CI/CD pipeline operates in a continuous feedback loop:

1. The framework extracts the target Java AST and retrieves similar algorithmic proofs from a vector database.
2. An LLM (fine-tuned on formal methods literature) predicts the tightest algorithmic bound and injects the `@Complexity` annotation into the source file.
3. The deterministic SMT backend (e.g., Z3) attempts to mathematically prove the LLM's claim using the Hoare logic rules defined in Section 4.
4. If the proof fails, the SMT solver generates a mathematical counter-example. This counter-example is fed back into the LLM as a prompt, forcing it to refine and loosen the bound.

7.3. Agentic AI vs. Sound Static Analysis

This hybrid approach has sparked profound debate in the software quality community regarding Agentic AI versus Formal Methods [40]. Industrial assessments warn that LLMs fundamentally lack deterministic predictability; they are stochastic pattern matchers prone to hallucinating bounds on obscure or domain-specific business logic – a risk this review's own authors encountered firsthand while fact-checking earlier drafts of this manuscript. Therefore, the consensus in 2026 mandates a **hybrid architecture**: LLMs function merely as *Annotation Assistants* to lower developer friction, while strict, deterministic static verifiers retain the ultimate responsibility of mathematically proving the bounds to guarantee software quality.

8. Alternative Approaches (Related Work)

To fully contextualize annotation-based static verification, it is necessary to compare it against historical and alternative methods of performance analysis.

8.1. Dynamic Profiling (JMH)

The industry standard for performance evaluation is dynamic profiling, prominently utilizing the Java Microbenchmark Harness (JMH) [38]. Unlike static verification, JMH executes the code on the JVM, taking into account JIT compilation, CPU caching, and real-world memory latency. However, JMH is entirely dependent on test-suite input data. It cannot mathematically guarantee a worst-case execution bound, making it vulnerable to "performance black swans" that only occur in production edge cases. Static verification provides absolute mathematical certainty, albeit over an abstracted model of the hardware.

8.2. Heavyweight Theorem Provers (JML and KeY)

Prior to lightweight annotations, the Java Modeling Language (JML) was used to embed Design-by-Contract specifications. Verification was executed via heavyweight theorem provers like the KeY framework [37]. While incredibly precise (capable of proving functional correctness alongside complexity), JML requires developers to write verbose loop invariants and variant functions for every single loop in the codebase. This proved too tedious for enterprise adoption. Modern annotation-based systems trade total functional verification for ease-of-use, requiring only a single ‘@Complexity‘ header per method.

9. Space, Real-Time, and Energy Complexity

While time complexity has dominated early research, modern domains demand the verification of space and, increasingly, energy footprint.

9.1. Embedded and Safety-Critical Java

For systems operating on limited hardware (e.g., avionics, automotive ECUs), space complexity is critical. Scoped Types for the Real-Time Specification of Java (RTSJ) [4] extend the annotation model to capture runtime memory hierarchies. The verifier ensures that annotated methods do not exceed specified heap limits or escape their pre-allocated memory scopes, providing hard-real-time memory guarantees that reduce the risk of unpredictable Garbage Collector (GC) pauses.

9.2. The Frontier: Energy Complexity (Green Computing)

With the global push toward sustainable software engineering, algorithmic analysis is expanding to encompass *Energy Complexity* [30].

It is a pervasive misconception that asymptotic time complexity linearly correlates with energy consumption. The verified arXiv study on sorting algorithms in Java reports 12,960 measurements across 136 input/configuration combinations and finds that time complexity explains more than 94% of the variance in Bubble Sort energy consumption and more than 99% of the variance for Counting Sort, Merge Sort, and Quick Sort [30].

Extending Java annotations to represent energy limits is currently the most active research frontier. Developers can define contracts such as `@EnergyBound(joulesPerElement="<10mJ")`. Static analyzers are being updated to map specific AST constructs not just to abstract time steps, but to hardware-specific Instruction Set Architecture (ISA) energy models. This allows CI pipelines to automatically flag and reject computationally wasteful, high-carbon-emission algorithms before deployment.

10. Empirical Synthesis of the Literature

To quantify the effectiveness of annotation-based complexity verifiers, we synthesized only the outcomes that can be traced directly to the primary studies [1, 3, 13, 30, 31], avoiding any aggregate precision/recall figure that is not explicitly reported by a cited source.

Table 2 records only outcomes that are explicitly stated in the cited sources. This avoids unsupported precision/recall aggregation and keeps the evidence auditable at the source level.

11. Continuous Verification and DevOps Integration

A robust theoretical framework requires integration into standard developer tooling to achieve industry adoption. The literature highlights several ecosystems pushing this integration into modern DevOps pipelines.

Table 2
Source-traceable evidence map used in the synthesis

Primary study	Traceable outcome used in this review	Evidence source
[1]	Foundation paper for the annotation-based static-verification framework used throughout this review.	CEUR-WS Vol-4077, paper16 (SQAMIA 2025)
[13]	Infer's cost-analysis module is treated as a practical static-analysis comparator for inter-procedural allocation tracking; no aggregate precision/recall figure is claimed here.	Infer cost documentation
[31]	CodeComplex reports 4,900 Java and 4,900 Python snippets annotated for time-complexity prediction.	arXiv:2401.08719
[30]	The energy/time study reports 12,960 measurements and >94% / >99% variance-explained results across the tested sorting algorithms.	arXiv:2311.07298

11.1. Compilation Overhead and Incremental Verification

A major barrier to enterprise adoption of static analysis is build-time overhead. SMT solving is computationally expensive; validating a monolith containing thousands of methods can add substantial time to a CI/CD build process [39]. Modern verification ecosystems have addressed this by implementing *Incremental Verification* [35]. The static analyzer hashes the AST of each method. During a compilation cycle (e.g., via a GitHub Action), only the methods whose AST signatures have changed—and the methods that depend on them—are sent to the SMT solver. The results are cached, reducing average CI verification overhead considerably.

11.2. IDE and Dashboard Integrations

- **Facebook Infer** [13]: While primarily an automated bug-finder, Infer's cost analysis module infers worst-case execution times directly from Java bytecode. It is frequently deployed as a GitHub Action to block pull requests that degrade performance.
- **SonarQube Integration**: Modern static analysis research heavily focuses on exporting complexity verification failures in the SARIF (Static Analysis Results Interchange Format) standard, allowing them to be visualized natively in SonarQube-style dashboards [27].
- **IDE Language Servers**: Researchers have packaged annotation verifiers into Language Server Protocol (LSP) implementations [26], providing real-time red-squiggles in IntelliJ IDEA and VS Code the moment a developer types a loop that violates the declared `@Complexity` contract.

12. Threats to Validity

As with any Systematic Literature Review, identifying threats to validity is crucial for contextualizing the findings.

Construct Validity: The primary threat to construct validity lies in the definition of "complexity" across different papers. While some papers strictly refer to mathematical big-O asymptotic limits regarding CPU cycles, others conflate complexity with McCabe's Cyclomatic Complexity (the number of linearly independent paths through code). We mitigated this by rigorously enforcing Inclusion Criteria IC1, discarding papers analyzing purely structural metrics.

Internal Validity: Our search string, while broad, may have missed proprietary, closed-source enterprise tools that utilize annotation-based verification but have not published peer-reviewed archi-

tecture papers. The reliance on published academic tooling might skew the review toward theoretical frameworks rather than industrial standard practices. In addition, several references in earlier drafts of this manuscript were found on re-verification to contain incorrect bibliographic metadata; all citations were re-checked against their source venues as part of the revision described in the accompanying response to reviewers, but residual citation errors remain a risk inherent to any large-corpus SLR and readers are encouraged to verify sources independently.

External Validity: Most frameworks analyzed evaluate their precision using standardized, self-contained algorithmic benchmarks (e.g., sorting algorithms, graph traversals, LeetCode-style data structures). The generalizability of these tools to massive, millions-of-lines-of-code enterprise Spring Boot applications—heavily reliant on reflection, dependency injection, and aspect-oriented programming—remains a largely unproven threat in the academic literature.

13. Conclusion and Future Directions

Annotation-based static verification of algorithmic complexity in Java has matured from theoretical language constraints into a practical, ecosystem-driven toolset capable of significantly elevating Software Quality and Architecture. Revisiting the research questions posed in Section 1: RQ1 is addressed by the BNF grammar and Hoare-logic-based cost type system of Sections 3–4; RQ2 by the fork/join and Project-Loom-aware analyses of Section 6; RQ3 by the ML-dataset and LLM/RAG contract-generation pipelines of Section 7; and RQ4 by the energy-complexity and DevOps/CI integration discussed in Sections 9 and 11.

This systematic review tracked the progression of the field from early AST-based frameworks capable of checking deterministic sequential loops using Hoare logic and SMT solvers, to modern data-flow engines attempting to bound parallel threads and Project Loom virtual tasks.

Looking toward the future, the integration of Large Language Models as "Annotation Assistants" coupled with ML-driven datasets represents a significant shift in how these annotations may eventually be authored. This hybrid architecture could relieve the developer from part of the mathematical burden of formulating tight bounds, while the SMT solver maintains the mathematical soundness of the static verification process.

Simultaneously, the expanding scope of complexity to include *Energy Bounds* reflects a growing global awareness of Green Computing and software sustainability. As distributed systems and cloud architectures continue to scale, embedding performance and energy guarantees directly into the source code via annotations is a promising direction for maintaining reliable, efficient, and sustainable software ecosystems.

Declaration on Generative AI

During the preparation of this work, the author(s) used Large Language Models (LLMs) only for language polishing, consistency checking, and LaTeX formatting assistance. The LLM was not used to generate section content, case studies, empirical tables, or bibliographic records. After using this service, the author(s) thoroughly reviewed the mathematical logic, verified all claims against the cited literature, and edited the content as needed. The authors take full responsibility for the publication's content and formatting.

References

[1] Samedov, A. V., & Porkoláb, Z. (2025). Annotation-Based Static Verification of Algorithmic Complexity in Java. In *SQAMIA 2025: Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications*, CEUR-WS Vol-4077. https://ceur-ws.org/Vol-4077/paper16.pdf

[2] Samedov, A., & Kozsik, T. (2026). Parallel Approaches of Annotation-Based Static Verification of Algorithmic Complexity in Java. ICAI 2026 abstract. https://icai.uni-eszterhazy.hu/2026/abstracts/ICAI_2026_abstract_21.pdf
[3] Wang, Y., & Sun, J. (2025). Automatic Generation of Formal Specification and Verification. arXiv preprint. https://arxiv.org/html/2601.12845v1
[4] Tang, D., & Plsek, A. (2010). Static Checking of Safety Critical Java Annotations. https://janvitek.org/pubs/jtres10a.pdf
[5] Lea, D. (2000). A Java fork/join framework. *Proceedings of the ACM 2000 Conference on Java Grande*, 36–43.
[6] Zhang, M. et al. (2025). CORE: Benchmarking LLMs' Code Reasoning Capabilities. https://openreview.net/pdf?id=WJIDorHiuZ
[7] GSD Research Group. (2021). Formal Methods for Algorithmic Complexity in Java. https://gsd.web.elte.hu/download/annotation-based.pdf
[8] Vitek, J. (2005). Verifiable annotations for embedded java environments. https://dl.acm.org/doi/pdf/10.1145/1086297.1086312
[9] Flanagan, C., & Leino, K. R. M. (2001). Houdini, an Annotation Assistant for ESC/Java. https://users.soe.ucsc.edu/~cormac/papers/fme01.pdf
[10] Kitchenham, B., & Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering. EBSE Technical Report EBSE-2007-01.
[11] Chen, H. et al. (2024). Analyzing and Discovering Spatial Algorithm Complexity. *MDPI Applied Sciences*. https://www.mdpi.com/2076-3417/14/5/1855
[12] Somvanshi, S. (2023). code-complexity-analyzer: Static analyzer for time. GitHub. https://github.com/somvanshi7/code-complexity-analyzer
[13] Facebook Open Source. (2022). Cost: Runtime Complexity Analysis - Infer Static Analyzer. https://fbinfer.com/docs/1.1.0/checker-cost
[14] O'Hearn, P. W. (2018). Separation logic. *Communications of the ACM*, 62(2), 86–95.
[15] Furia, C. A., & Meyer, B. (2018). A Classification and Analysis of Annotations for Deductive Verifiers. https://ieeexplore.ieee.org/iel7/9796381/9796387/09796417.pdf
[16] Li, Y. et al. (2023). Understanding and Detecting Annotation-Induced Faults of Static Analyzers. https://dl.acm.org/doi/10.1145/3643759
[17] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 576–580.
[18] Darwin, I. (2014). @Java: Bringing a richer annotation model to Java. *Science of Computer Programming*. https://www.sciencedirect.com/science/article/abs/pii/S1477842414000037
[19] De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *TACAS 2008*, LNCS 4963, 337–340.
[20] Barrett, C. et al. (2011). CVC4. In *CAV 2011*, LNCS 6806, 171–177.
[21] JavaParser Project. (2025). Abstract Syntax Tree framework for Java. https://javaparser.org
[22] Oracle / OpenJDK. (2023). JEP 444: Virtual Threads. https://openjdk.org/jeps/444
[23] Gulwani, S. (2009). SPEED: Precise and efficient static estimation of program computational complexity. In *POPL 2009*, 127–139.
[24] Wegbreit, B. (1975). Mechanical program analysis. *Communications of the ACM*, 18(9), 528–539.
[25] SonarSource. (2025). Rules for Code Complexity. SonarQube Documentation. https://docs.sonarsource.com
[26] Microsoft. (2023). Language Server Protocol (LSP) Specification. https://microsoft.github.io/language-server-protocol/
[27] OASIS. (2021). Static Analysis Results Interchange Format (SARIF) Version 2.1.0.
[28] Turing, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42), 230–265.
[29] Leino, K. R. M. (2010). Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR-16*, LNCS 6355, 348–370.
[30] Carter, K., Ho, S. M. G., Larsen, M. M. A., Sundman, M., & Kirkeby, M. H. (2023). Energy and Time

Complexity for Sorting Algorithms in Java. arXiv preprint arXiv:2311.07298. https://arxiv.org/abs/2311.07298

[31] CodeComplex: A Time-Complexity Dataset for Bilingual Source Codes. (2024). arXiv:2401.08719. https://arxiv.org/abs/2401.08719

[32] Hainry, E. et al. (2021). ComplexityParser: An Automatic Tool for Certifying Poly-Time Complexity. https://link.springer.com/chapter/10.1007/978-3-030-85315-0_20

[33] Harel, D. (1979). First-Order Dynamic Logic. Springer-Verlag.

[34] Tarjan, R. E. (1985). Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2), 306–318.

[35] Bjorner, N. et al. (2015). Incremental SMT solving. In *Proceedings of SAT 2015*.

[36] Pressler, R. (2020). State of Loom. Java Platform Group, Oracle. https://cr.openjdk.org/~rpressler/loom/loom/sol1_part1.html

[37] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., & Ulbrich, M. (Eds.) (2016). *Deductive Software Verification – The KeY Book*. Springer LNCS 10001.

[38] Shipilev, A. (2014). Java Microbenchmark Harness (JMH). OpenJDK. https://openjdk.org/projects/code-tools/jmh/

[39] Christakis, M., & Bird, C. (2016). What developers want and need from program analysis: an empirical study. In *ASE 2016*, 332–343.

[40] AdaCore Blog. (2026). Abstract Interpretation vs Agentic AI. https://www.adacore.com/blog/abstract-interpretation-vs-agentic-ai