Deterministic Code Auditing for Production-Ready Software

A Polyglot Static Analysis System with Fruition Scoring

David Jean Charlot, PhD Open Interface Engineering, Inc. (openIE) University of California, Santa Barbara (UCSB) david@openie.dev | dcharlot@ucsb.edu

February 2026 | Version 1.0

Abstract

Current AI-assisted code review relies on large language models that *guess* whether code is correct—a fundamentally flawed approach that produces inconsistent, non-reproducible, and often hallucinated assessments. We present **AI-Verify**, a deterministic code auditing system that answers a simple question: *is this code production-ready?* Rather than inferring code quality through probabilistic generation, AI-Verify applies finite, enumerable static analysis rules via tree-sitter parsing across 15 programming languages, producing identical results for identical inputs with zero hallucination risk.

We introduce the **Fruition Score**—a composite metric quantifying code completeness based on detection of incomplete implementations (TODO, FIXME, unimplemented!), fake/placeholder values, undocumented unsafe blocks, and code complexity. Our polyglot rule engine supports language-specific patterns while maintaining cross-language semantic consistency. Experimental evaluation on 50,000 files across 5 languages demonstrates 94.7% precision in detecting incomplete code, with analysis completing in <100ms per file.

The key insight: **code quality verification does not require inference**. Unlike code generation where the output space is vast, verification operates on a finite set of checkable properties. AI-Verify embodies the wisdom principle—*don't guess if code is correct; verify it deterministically*.

Keywords: static analysis, code quality, deterministic verification, tree-sitter, polyglot programming, software metrics

---

1. Introduction

1.1 The Problem with AI Code Review

Modern AI code review tools suffer from a fundamental flaw: they *infer* code quality rather than *verify* it. Consider a typical LLM-based code review:

```
Input: "Review this function for issues" Output: "This code looks good overall. The error
handling could be improved..."
```

This response exhibits several problems: 1. **Non-determinism**: The same code may receive different reviews on different invocations 2. **Hallucination**: The model may fabricate issues that don't exist or miss real problems 3. **Vagueness**: "Looks good" and "could be improved" are not actionable metrics 4. **Inconsistency**: Review quality varies with prompt phrasing and context window

1.2 Our Thesis

We argue that code quality verification is a fundamentally different problem than code generation, and **does not require inference**:

> ***Code generation***: *Vast output space, requires creativity, benefits from probabilistic sampling* ***Code verification***: *Finite property space, requires precision, demands deterministic checking*

The set of properties that make code "production-ready" is enumerable: - No incomplete implementations (TODO, FIXME, unimplemented) - No placeholder/fake values (fake_*, hardcoded test data) - Safe code or documented unsafe blocks - Bounded complexity (cognitive load, cyclomatic complexity) - No known vulnerability patterns

Each property is deterministically checkable. There is no need to *guess* whether code contains a TODO—we can *verify* it with a finite-state parser.

1.3 The Wisdom Principle Applied to Verification

This work is part of a broader thesis on computational wisdom:

> ***Current AI is simulated intelligence***: *inference without discernment, guessing even when knowing is possible.* ***The next generation of AI is wisdom***: *knowing when NOT to*

> *infer*.

For code verification, the wisdom principle is absolute: **never infer what you can verify**. Every property we check has a deterministic answer. Inference would be not just unnecessary but actively harmful—introducing non-determinism into a domain that demands consistency.

1.4 Contributions

This paper makes the following contributions:

1. **Fruition Score**: A composite metric quantifying code production-readiness (0-100%)
2. **Polyglot Rule Engine**: Language-agnostic patterns with language-specific adaptations across 15 languages
3. **Tree-sitter Integration**: Deterministic AST parsing enabling precise pattern matching
4. **Empirical Validation**: 94.7% precision on incomplete code detection across 50,000 files
5. **Energy Efficiency**: 1000× lower energy than LLM-based review with higher consistency

---

2. Background and Related Work

2.1 Traditional Static Analysis

Traditional static analysis tools have established the foundation for deterministic code verification:

Linters: - ESLint [1] for JavaScript: Rule-based checking with configurable severity - Clippy [2] for Rust: Compiler-integrated lint passes - Pylint [3] for Python: Style and error detection

Security Scanners: - Semgrep [4]: Pattern-based security scanning - CodeQL [5]: Query-based vulnerability detection - Bandit [6]: Python security linter

Limitations of existing tools: - Language-specific implementations requiring separate toolchains - Focus on syntax/style rather than semantic completeness - No unified "production-readiness" metric - Limited cross-language consistency

2.2 AI-Based Code Review

Recent work has applied LLMs to code review:

- **GitHub Copilot Code Review** [7]: GPT-based review suggestions
- **Amazon CodeGuru** [8]: ML-based code recommendations
- **DeepCode/Snyk Code** [9]: Neural pattern matching for vulnerabilities

These systems achieve impressive results on certain tasks but suffer from the fundamental inference problem: non-determinism, hallucination, and inconsistent severity assessment.

2.3 Tree-sitter and Universal Parsing

Tree-sitter [10] provides incremental parsing infrastructure with key properties: - **Determinism**: Same input → same AST, always - **Polyglot**: Grammars available for 40+ languages - **Performance**: Sub-millisecond parsing for typical files - **Error tolerance**: Produces partial ASTs for malformed input

Tree-sitter enables our polyglot approach: write semantic patterns once, apply across languages through normalized AST queries.

2.4 Positioning

System	Deterministic	Polyglot	Completeness	Energy
ESLint	Yes	No (JS only)	Style focus	Low
Clippy	Yes	No (Rust only)	Rust-specific	Low
Semgrep	Yes	Yes (patterns)	Security focus	Low
Copilot Review	No	Yes	General	High
CodeGuru	No	Limited	ML-based	High
AI-Verify	**Yes**	**Yes (15 langs)**	**Fruition focus**	**Low**

AI-Verify uniquely combines deterministic analysis with polyglot support and production-readiness focus.

---

3. The Fruition Score

3.1 Definition

Definition 3.1 (Fruition Score): The fruition score $F \in [0, 100]$ quantifies code production-readiness as:

$$F = 100 \times \left(1 - \frac{\sum_{i} w_i \cdot c_i}{\sum_{i} w_i \cdot m_i}\right)$$

where: - c_i = count of issues in category i - m_i = maximum expected issues (normalization factor) - w_i = weight for category i

3.2 Issue Categories

Category	Weight	Description	Detection Method
Incomplete	1.0	TODO, FIXME, unimplemented	Keyword + AST pattern
Fake Values	0.8	fake_*, placeholder, hardcoded test data	Identifier + literal analysis
Unsafe Code	0.6	Undocumented unsafe blocks	AST + comment analysis
Complexity	0.4	High cyclomatic/cognitive complexity	Control flow analysis
Dead Code	0.3	Unreachable statements	Dataflow analysis

3.3 Interpretation

Score Range	Interpretation	Recommendation
90-100	Production-ready	Ship with confidence
80-89	Near-complete	Minor cleanup needed
60-79	Work in progress	Address TODOs before merge

40-59	Early development	Not ready for review
0-39	Prototype/Sketch	Significant work remaining

3.4 Score Properties

Theorem 3.1 (Determinism): For any code C, the fruition score F(C) is deterministic:

$$\forall C: F(C)_1 = F(C)_2 = ... = F(C)_n$$

Proof: Each component of F is computed via deterministic tree-sitter parsing and finite pattern matching. No probabilistic sampling is involved. ■

Theorem 3.2 (Monotonicity): Fixing an issue never decreases the score:

$$\text{fix}(C, i) \implies F(\text{fix}(C, i)) \geq F(C)$$

Proof: Each issue contributes a positive term to the penalty. Removing an issue removes its penalty contribution. ■

---

4. Architecture

4.1 System Overview

```
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
AI-Verify Pipeline ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
Layer 1: Polyglot Parsing (Tree-sitter) ■ ■ • 15 language grammars ■ ■ • Deterministic AST
generation ■ ■ • Error-tolerant parsing ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
Layer 2: Pattern Matching Engine ■ ■ • Language-agnostic semantic patterns ■ ■ •
Language-specific syntax adaptations ■ ■ • Comment and string analysis ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
Layer 3: Metric Computation ■ ■ • Cyclomatic complexity ■ ■ • Cognitive complexity ■ ■ •
Halstead metrics ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
Layer 4: Fruition Scoring ■ ■ • Issue aggregation ■ ■ • Weighted combination ■ ■ • Report
generation (JSON, SARIF) ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

4.2 Supported Languages

Language	Grammar	Incomplete	Fake	Unsafe	Complexity
Rust	tree-sitter-rust	✓	✓	✓	✓
Python	tree-sitter-python	✓	✓	N/A	✓
JavaScript	tree-sitter-javascript	✓	✓	N/A	✓
TypeScript	tree-sitter-typescript	✓	✓	N/A	✓
Go	tree-sitter-go	✓	✓	✓	✓
Java	tree-sitter-java	✓	✓	N/A	✓
C	tree-sitter-c	✓	✓	✓	✓
C++	tree-sitter-cpp	✓	✓	✓	✓
C#	tree-sitter-c-sharp	✓	✓	✓	✓
Ruby	tree-sitter-ruby	✓	✓	N/A	✓
PHP	tree-sitter-php	✓	✓	N/A	✓
Bash	tree-sitter-bash	✓	✓	N/A	✓
Haskell	tree-sitter-haskell	✓	✓	✓	✓
OCaml	tree-sitter-ocaml	✓	✓	✓	✓
Scala	tree-sitter-scala	✓	✓	N/A	✓

4.3 Rule Engine Design

Pattern Specification:

```
pub struct Rule { pub id: String, // e.g., "INCOMPLETE_TODO" pub severity: Severity, //
Error, Warning, Info pub category: Category, // Incomplete, Fake, Unsafe, Complexity pub
languages: Vec<Lang>, // Which languages apply pub pattern: Pattern, // AST pattern or regex
pub message: String, // Human-readable description pub suggestion: Option<String>, //
Auto-fix suggestion }
```

Language Adaptation:

```
// Semantic pattern: "incomplete implementation marker" // Adapts to language-specific syntax: // Rust: todo!(), unimplemented!(), panic!("not implemented") // Python: raise NotImplementedError, pass # TODO // JavaScript: throw new Error("not implemented") // Go: panic("not implemented")
```

---

5. Detection Algorithms

5.1 Incomplete Code Detection

Definition 5.1 (Incomplete Code): Code is incomplete if it contains explicit markers indicating unfinished implementation.

Markers by category:

Category	Patterns	Languages
Comment markers	TODO, FIXME, XXX, HACK, BUG	All
Macro markers	todo!(), unimplemented!()	Rust
Exception markers	NotImplementedError, NotImplementedException	Python, C#, Java
Panic markers	panic!("not implemented"), panic("TODO")	Rust, Go
Empty bodies	pass, ... (ellipsis), {} with only comments	Python, Rust

Algorithm 5.1 (Incomplete Detection):

```
Input: AST node n, comments C Output: List of incomplete findings 1. For each comment c in C: - If c matches /TODO|FIXME|XXX|HACK/i: - Emit finding at c.location 2. For each function body b in AST: - If b contains only: pass, ..., todo!(), unimplemented!(): - Emit finding at b.location 3. For each macro invocation m: - If m.name in {todo, unimplemented, panic} and m.args matches "not implemented|TODO": - Emit finding at m.location
```

5.2 Fake Value Detection

Definition 5.2 (Fake Value): A value that is clearly placeholder data not suitable for production.

Patterns: - Identifier patterns: `fake_*`, `test_*`, `dummy_*`, `placeholder_*`, `mock_*` - Literal patterns: `"lorem ipsum"`, `"test"`, `12345`, `"password"`, `"xxx"` - Magic numbers: hardcoded constants without const declaration

Algorithm 5.2 (Fake Value Detection):

```
Input: AST Output: List of fake value findings 1. For each identifier i: - If i.name matches
/^(fake|test|dummy|placeholder|mock|tmp|temp)_/i: - Emit finding at i.location with
confidence HIGH 2. For each string literal s: - If s.value in KNOWN_FAKE_STRINGS: - Emit
finding at s.location with confidence MEDIUM - If s.value matches
/^(test|fake|xxx+|placeholder)<em>/i: - Emit finding at s.location with confidence HIGH 3.
For each numeric literal n not in const declaration: - If n.value in MAGIC_NUMBERS: - Emit
finding at n.location with confidence LOW
```

5.3 Unsafe Code Detection

Definition 5.3 (Undocumented Unsafe): Unsafe code blocks without SAFETY documentation explaining the invariants maintained.

Language-specific unsafe constructs: - **Rust**: `unsafe { }` blocks, `unsafe fn`, `unsafe impl` - **C/C++**: Pointer arithmetic, `reinterpret_cast`, raw memory operations - **Go**: `unsafe.Pointer`, `//go:nosplit`, `//go:noescape` - **C#**: `unsafe { }` blocks, `fixed` statements

Algorithm 5.3 (Unsafe Detection):

```
Input: AST, Comments Output: List of unsafe findings 1. For each unsafe block/construct u: -
preceding_comments = comments_before(u, 3 lines) - If not any(c contains "SAFETY" for c in
preceding_comments): - Emit finding: "Unsafe block without SAFETY documentation" - Else: -
Verify SAFETY comment is non-empty
```

5.4 Complexity Analysis

Cyclomatic Complexity [11]:

<div class="math-block">V(G) = E - N + 2P</div>

where E = edges, N = nodes, P = connected components in control flow graph.

Cognitive Complexity [12]: Increments for: - Control flow breaks (if, for, while, switch): +1 - Nesting: +1 per level - Boolean operators in conditions: +1 each - Recursion: +1

Thresholds: | Complexity | Cyclomatic | Cognitive | |------------|------------|-----------| | Low | 1-10 | 1-5 | | Medium | 11-20 | 6-15 | | High | 21-50 | 16-30 | | Very High | >50 | >30 |

---

6. Implementation

6.1 Core Data Structures

```
/// Audit finding from analysis #[derive(Debug, Clone, Serialize)] pub struct Finding { pub severity: Severity, pub rule_id: String, pub message: String, pub location: Location, pub suggestion: Option&lt;String&gt;, } /// File-level audit report #[derive(Debug, Clone, Serialize)] pub struct FileReport { pub path: PathBuf, pub language: Language, pub lines: usize, pub findings: Vec&lt;Finding&gt;, pub metrics: Metrics, } /// Project-level statistics #[derive(Debug, Clone, Serialize)] pub struct ProjectStats { pub files_analyzed: usize, pub total_lines: usize, pub incomplete_count: usize, pub fake_count: usize, pub unsafe_count: usize, pub fruition_score: f64, pub details: Vec&lt;FileReport&gt;, }
```

6.2 Tree-sitter Integration

```
/// Universal AST walker for pattern matching pub fn walk_ast&lt;F&gt;(node: Node, source: &amp;[u8], visitor: &amp;mut F) where F: FnMut(Node, &amp;str), { let mut cursor = node.walk(); loop { let current = cursor.node(); let text = current.utf8_text(source).unwrap_or(""); visitor(current, text); if cursor.goto_first_child() { continue; } while !cursor.goto_next_sibling() { if !cursor.goto_parent() { return; } } } } /// Language-agnostic incomplete detection pub fn detect_incomplete(ast: &amp;Tree, source: &amp;[u8], lang: Language) -&gt; Vec&lt;Finding&gt; { let mut findings = Vec::new(); walk_ast(ast.root_node(), source, &amp;mut |node, text| { match node.kind() { "comment" =&gt; { if INCOMPLETE_REGEX.is_match(text) { findings.push(Finding { severity: Severity::Warning, rule_id: "INCOMPLETE_COMMENT".into(), message: format!("Incomplete marker in comment: {}", text.trim()), location: node.into(), suggestion: Some("Implement or remove TODO".into()), }); } } "macro_invocation" if lang == Language::Rust =&gt; { if text.starts_with("todo!") || text.starts_with("unimplemented!") { findings.push(Finding { severity: Severity::Error, rule_id: "INCOMPLETE_MACRO".into(), message: "Incomplete implementation macro".into(), location: node.into(), suggestion: Some("Implement the function body".into()), }); } } _ =&gt; {} } }); findings }
```

6.3 SARIF Output

AI-Verify produces SARIF (Static Analysis Results Interchange Format) [13] output for integration with standard tooling:

```
{ "</em>schema":
"https://raw.githubusercontent.com/oasis-tcs/sarif-spec/master/Schemata/sarif-schema-2.1.0.json",
"version": "2.1.0", "runs": [{ "tool": { "driver": { "name": "AI-Verify", "version": "1.0.0",
```

```
"rules": [...] } }, "results": [ { "ruleId": "INCOMPLETE_TODO", "level": "warning",
"message": { "text": "TODO marker found" }, "locations": [{ "physicalLocation": {
"artifactLocation": { "uri": "src/lib.rs" }, "region": { "startLine": 42 } } }] } ] }] }
```

7. Experimental Evaluation

7.1 Methodology

Corpus: 50,000 files across 5 languages from GitHub's top-starred repositories

Language	Files	Lines	Repositories
Rust	12,000	2.4M	500
Python	15,000	3.1M	600
JavaScript	10,000	1.8M	400
Go	8,000	1.6M	350
Java	5,000	1.2M	250

Ground Truth: 2,000 files manually labeled for incomplete code, fake values, and unsafe patterns by 3 annotators with >90% inter-annotator agreement.

7.2 Detection Accuracy

Category	Precision	Recall	F1
Incomplete (TODO/FIXME)	98.2%	96.8%	97.5%
Incomplete (macro/exception)	94.1%	91.3%	92.7%
Fake Values (identifiers)	89.4%	85.2%	87.2%

Fake Values (literals)	76.3%	72.1%	74.1%
Unsafe (undocumented)	92.8%	88.4%	90.5%
Overall	**94.7%**	**91.2%**	**92.9%**

Key observations: - Comment-based TODO detection achieves near-perfect precision - Fake value detection has lower precision due to legitimate test files - Unsafe detection benefits from clear SAFETY convention in Rust ecosystem

7.3 Performance

Metric	Value
Parsing time (median)	2.3ms/file
Analysis time (median)	12ms/file
Total time (median)	14.5ms/file
Memory usage	45MB base + 1KB/file
Throughput	68 files/second

Comparison with LLM-based review:

Approach	Time/file	Energy/file	Deterministic
GPT-4 Review	3-8 seconds	~50 Joules	No
Claude Review	2-5 seconds	~35 Joules	No
AI-Verify	15ms	~0.05 Joules	**Yes**

Energy savings: 700-1000× compared to LLM-based review.

7.4 Fruition Score Distribution

Analysis of 10,000 production files vs. 5,000 work-in-progress files:

Score Range	Production (%)	WIP (%)
90-100	72.3%	8.4%
80-89	18.1%	15.2%
60-79	7.2%	31.7%
40-59	1.8%	28.4%
0-39	0.6%	16.3%

The fruition score effectively discriminates between production-ready and in-progress code.

7.5 Cross-Language Consistency

Testing the same semantic patterns across language implementations:

Pattern	Rust	Python	JS	Go	Variance
TODO detection	97.8%	98.1%	97.5%	98.0%	0.23%
Empty body	95.2%	94.8%	93.1%	95.5%	0.92%
Fake identifier	88.9%	89.3%	87.2%	88.5%	0.82%

Low variance confirms cross-language semantic consistency.

---

8. Integration with AI-Author and AI-Compile

8.1 The Deterministic Code Lifecycle

AI-Verify is part of a three-product suite embodying the wisdom principle:

```
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
WISDOM-BASED AI CODE LIFECYCLE ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
AI-Author: Generate code ■ ■ • Don't infer implementations → LOOK THEM UP ■ ■ • Pattern
database + LSP + deterministic search ■ ■ • Only use LLM for genuinely novel problems ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
AI-Verify: Validate code ■ ■ • Don't infer correctness → VERIFY DETERMINISTICALLY ■ ■ •
Tree-sitter parsing + finite rule engine ■ ■ • Never use inference for verification ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ ■
AI-Compile: Execute code ■ ■ • Don't infer results → COMPILE AND RUN ■ ■ • Polyglot lifting
+ JIT compilation + GPU dispatch ■ ■ • Deterministic execution, neural context tracking ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

8.2 Verification in the Generation Loop

AI-Verify integrates with AI-Author to verify generated code before acceptance:

```
// AI-Author generates code let generated = ai_author.generate(signature, context).await?; //
AI-Verify validates before returning let audit = ai_verify.audit(&generated).await?; if
audit.fruition_score < 80.0 { // Reject incomplete code return
Err(GenerationError::IncompleteCode { score: audit.fruition_score, issues: audit.findings,
}); } // Only return production-ready code Ok(generated)
```

This creates a **verification-generation duality**: AI-Author generates candidates, AI-Verify filters to production-ready output. The combination achieves higher quality than either alone.

8.3 Continuous Verification

AI-Verify integrates into CI/CD pipelines:

```
# GitHub Actions workflow - name: AI-Verify Audit run: | ai-verify . --format sarif --output
results.sarif if [ $(jq '.runs[0].results | length' results.sarif) -gt 0 ]; then echo
"::error::AI-Verify found issues" exit 1 fi - name: Upload SARIF uses:
github/codeql-action/upload-sarif@v2 with: sarif_file: results.sarif
```

---

9. Discussion

9.1 Why Determinism Matters

Non-deterministic code review creates several problems:

1. **Review roulette**: Developers retry reviews hoping for better results

2. **Inconsistent standards**: Same code receives different feedback
3. **Trust erosion**: Developers dismiss AI review as unreliable
4. **Audit failure**: Cannot reproduce review results for compliance

Deterministic verification eliminates these issues. The same code always receives the same assessment. Results are reproducible, auditable, and trustworthy.

9.2 Limitations

False positives: - Test files legitimately contain "fake" values - TODO comments may be informational ("TODO list implementation") - Solution: Context-aware filtering, file path patterns

Language coverage: - Some languages lack mature tree-sitter grammars - Language-specific idioms may not translate universally - Solution: Incremental grammar development, language-specific rules

Semantic analysis: - Cannot detect semantic bugs (wrong algorithm, incorrect logic) - Cannot verify business rule compliance - Solution: Complement with testing, not replace

9.3 When Inference is Appropriate

AI-Verify takes a hard stance: **never infer verification results**. However, inference has legitimate uses in the broader development lifecycle:

- **Code generation** (AI-Author): When no pattern exists, synthesis is appropriate
- **Documentation generation**: Natural language output benefits from generation
- **Code explanation**: Summarizing code for humans is inherently generative

The key is discernment: use the right tool for each task. Verification demands determinism; generation may benefit from inference.

---

10. Conclusion

We have presented AI-Verify, a deterministic code auditing system that answers the question "is this code production-ready?" without inference. Our key contributions:

1. **Fruition Score**: A composite metric quantifying code completeness (0-100%)

2. **Polyglot Analysis**: Consistent detection across 15 programming languages
3. **Deterministic Verification**: Same code → same results, always
4. **Energy Efficiency**: 1000× lower energy than LLM-based review
5. **Integration**: Seamless combination with AI-Author and AI-Compile

10.1 The Wisdom Principle for Verification

Code verification does not require inference. The properties that make code production-ready—completeness, safety, bounded complexity—are deterministically checkable. Using LLMs for verification introduces non-determinism into a domain that demands consistency.

> ***Don't guess if code is correct. Verify it.***

This is the application of computational wisdom to software quality: know when NOT to infer. For verification, the answer is always—inference is unnecessary, harmful, and wasteful.

10.2 Future Work

- **Custom rule authoring**: User-defined patterns and metrics
- **IDE integration**: Real-time fruition scoring during development
- **Historical analysis**: Track fruition score trends over repository history
- **Cross-project patterns**: Learn common issues from large-scale analysis

AI-Verify demonstrates that deterministic methods can match or exceed AI inference for well-defined tasks. The path forward is not bigger models but smarter architectures—systems wise enough to know when guessing is unnecessary.

References

[1] ESLint. "ESLint - Pluggable JavaScript Linter." https://eslint.org/

[2] Rust Clippy. "A collection of lints to catch common mistakes." https://github.com/rust-lang/rust-clippy

[3] Pylint. "Python static code analysis tool." https://pylint.org/

[4] Semgrep. "Lightweight static analysis for many languages." https://semgrep.dev/

[5] GitHub. "CodeQL: Discover vulnerabilities across a codebase." https://codeql.github.com/

[6] PyCQA. "Bandit: Security linter for Python." https://bandit.readthedocs.io/

[7] GitHub. "GitHub Copilot." https://github.com/features/copilot

[8] Amazon. "Amazon CodeGuru." https://aws.amazon.com/codeguru/

[9] Snyk. "Snyk Code: Real-time semantic code analysis." https://snyk.io/product/snyk-code/

[10] Tree-sitter. "An incremental parsing system." https://tree-sitter.github.io/tree-sitter/

[11] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, 1976. https://ieeexplore.ieee.org/document/1702388

[12] G. A. Campbell, "Cognitive Complexity: A new way of measuring understandability," *SonarSource*, 2018. https://www.sonarsource.com/resources/cognitive-complexity/

[13] OASIS. "Static Analysis Results Interchange Format (SARIF) Version 2.1.0." https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html

---

Appendix A: Rule Definitions

A.1 Incomplete Code Rules

```
- id: INCOMPLETE_TODO severity: Warning pattern: /\b(TODO|FIXME|XXX|HACK|BUG)\b/i scope:
comment message: "Incomplete marker in comment" - id: INCOMPLETE_UNIMPLEMENTED severity:
Error languages: [rust] pattern: macro_invocation[name=unimplemented|todo] message:
"Unimplemented macro" - id: INCOMPLETE_NOT_IMPLEMENTED severity: Error languages: [python]
pattern: raise_statement[exception=NotImplementedError] message: "NotImplementedError raised"
```

A.2 Fake Value Rules

```
- id: FAKE_IDENTIFIER severity: Warning pattern:
identifier[name=/^(fake|test|dummy|mock|placeholder)_/i] scope: non-test-file message:
"Placeholder identifier in non-test code" - id: FAKE_PASSWORD severity: Error pattern:
string_literal[value=/password|secret|apikey/i] message: "Potential hardcoded credential"
```

Contact: david@openie.dev | dcharlot@ucsb.edu **Latest Updates**: https://openie.dev/projects/ai-verify