WHITE PAPER

ARGUS

Automated Review & Grading Utility for Software

Build Plan, Evidence Architecture & Offline Deployment

Anna R. Dudley

Signal · Strategy · Systems

April 2026

Purpose

Deploy an offline-first agent that scans software repositories for evidence of feature implementation, producing a tiered confidence report. The agent runs at the edge with zero internet dependency, with optional LLM disambiguation when connectivity is available.

ARGUS is an evidence scanner, not a compliance certifier. It identifies structural indicators that a feature is present and classifies the strength of that evidence. Human reviewers use ARGUS output to prioritize where to focus their assessment, not to replace it.

How It Works

Users define required features in a YAML configuration file. Each feature specifies what to look for using a layered check architecture, progressing from surface-level indicators to deeper structural analysis:

Tier 1 — Presence: file existence patterns, dependency manifest lookups, documentation keyword searches.

Tier 2 — Usage: AST-based analysis confirming functions, classes, or modules are imported and actually invoked, not merely declared.

Tier 3 — Integration: call graph and data flow analysis verifying that identified functions are reachable from the relevant entry points or data paths. Reachability indicates potential integration but does not guarantee execution under real runtime conditions such as feature flags, configuration states, or environment constraints.

Tier 4 — Behavioral: inspection of existing test suites, CI configuration, and coverage reports to determine whether the feature is exercised by the project’s own tests. Tier 4 reflects the presence of test coverage related to the feature, not verified runtime correctness. Test quality and assertion depth are not evaluated.

The agent walks the target software directory, executes checks at each tier, and reports the highest evidence tier reached per feature. Results are output as both machine-readable JSON and a human-readable Markdown report.

Evidence Tier Model

ARGUS scores feature by the depth of evidence found, not by counting pattern matches. A single Tier 3 signal outweighs any number of Tier 1 signals.

Features that reach only Tier 1 are flagged as low-confidence and marked for human review. Features reaching Tier 3 or Tier 4 are reported as high confidence. This tiered model replaces flat weighted scoring, which conflates quantity of evidence with quality of evidence.

Evidence tier reflects structural depth of analysis (how deep into the codebase ARGUS can verify), while confidence reflects the likelihood that the detected structure corresponds to a valid implementation. A Tier 3 result with low confidence (e.g., due to ambiguous call graph resolution) is reported differently from a Tier 3 result with high confidence. This separation prevents conflating

depth of evidence with certainty of evidence.

Performance Evaluation

ARGUS performance is evaluated using precision and recall metrics across labeled datasets. Precision measures the rate of false positives (features incorrectly identified as present), while recall measures the rate of false negatives (features incorrectly identified as absent). F1 score provides a combined measure. Tier-specific accuracy is tracked to assess reliability at each evidence level. Evaluation datasets are composed of known implementations across supported languages, with cross-language performance comparisons to identify systematic gaps in language pack coverage.

Build Process

ONCE REQUIREMENTS ARE RECEIVED

Phase 1: Requirements Translation

Convert the user's feature requirements into the YAML configuration schema. Each requirement becomes a feature entry with checks defined at each applicable tier. For example, a requirement like "must support encrypted data at rest" translates to:

T1: file patterns for encryption modules, dependency checks for cryptography libraries.

T2: AST queries confirming encrypt/decrypt functions are called, not just imported.

T3: call graph verification that encryption calls are reachable from data write paths.

T4: test suite checks for encryption-related test cases and coverage.

Each feature must include a structured definition specifying scope, expected system boundaries, and acceptable implementation patterns to reduce variability in interpretation. Required fields include feature name, scope definition, expected entry points, and acceptable implementation patterns. Well-defined features produce consistent results across reviewers; poorly defined features introduce measurement noise that compounds across tiers.

Phase 2: Language Pack Selection

Select pre-built AST query patterns for the languages present in the target software. ARGUS uses tree-sitter for cross-language parsing, shipping language packs for Python, JavaScript/TypeScript, Go, Java, C#, Rust, and C/C++. Each pack includes common check patterns for standard features (authentication, encryption, logging, error handling). Users extend or override these patterns for domain-specific requirements.

This replaces the previous approach of requiring users to author raw regex patterns per language. Regex-based code matching is limited to Tier 1 checks only, as it cannot distinguish between code that exists and code that executes.

Phase 3: Calibration & Validation

Run the agent against known software samples where expected outcomes are already understood. Calibration now operates at two levels:

Pattern tuning: adjust AST queries and file patterns until Tier 1 and Tier 2 checks align with ground truth.

Integration validation: verify that call graph analysis correctly identifies connected vs. orphaned code paths.

Calibration samples must be diverse. Tuning against a narrow set of codebases produces an agent that only recognizes implementations structured like the calibration samples. Calibration datasets must include at least five structurally distinct implementations per feature, spanning multiple frameworks and architectural patterns, to reduce overfitting.

Ground Truth Definition

Ground truth labels are established via dual review by domain experts (senior engineers or subject matter specialists), with disagreements resolved through adjudication. Each feature is labeled as Present, Absent, or Ambiguous based on predefined criteria. Expected outcomes are derived from human-labeled datasets, reference implementations, and gold-standard repositories. Labeling criteria define what counts as “feature present” at each evidence tier.

Phase 4: Deployment Packaging

Select the runtime appropriate for the target environment: Python for rapid iteration or Go for a single compiled binary with zero runtime dependencies. Tree-sitter language grammars are bundled as shared libraries alongside the agent. Package the agent with its configuration file and language packs for distribution to edge nodes.

Phase 5: Optional LLM Disambiguation

When internet access is available, the agent can forward ambiguous findings to any OpenAI-compatible LLM API (Ollama, LM Studio, or cloud providers). The LLM is used strictly for disambiguation, not scoring:

Only PARTIAL-confidence results are sent to the LLM.

The LLM receives the specific ambiguous snippet and a binary question: does this code appear to implement the feature, or does it merely reference it?

The LLM’s answer flips the result to Meets or Does Not Meet Evidence Criteria. It does not blend a numeric score.

Clear Meets and Does Not Meet results from strong evidence never touch the LLM.

This ensures that offline and online runs produce identical results for unambiguous cases. Ambiguous cases are flagged as “needs review” in offline mode rather than silently scored differently.

LLM responses are recorded as advisory signals with associated confidence scores (0–1). Ambiguous results may remain classified as UNCERTAIN rather than forced into binary classification. LLM decisions are logged separately and do not overwrite original evidence tier assignments.

Deliverables

All findings include traceable evidence artifacts, enabling reviewers to independently verify each classification. Each result includes the exact file paths where evidence was found, AST match snippets showing the relevant code structure, and call graph traces demonstrating reachability. This audit trail ensures that no ARGUS result requires trust in a black-box process.

Known Limitations

ARGUS is a static evidence scanner. The following limitations are inherent to this approach and cannot be fully resolved without runtime analysis:

Behavioral properties are out of scope. Requirements like “must handle 1,000 concurrent users” or “must fail gracefully under network loss” describe runtime behavior. ARGUS can check whether relevant code structures exist (connection pools, retry logic, circuit breakers) but cannot verify they work correctly under load. Tier 4 evidence from test suites partially mitigates this, but only if the project’s tests cover these scenarios.

Correctness is not assessed. ARGUS determines whether code that appears to implement a feature is present and connected. It does not determine whether that code is correct. An encryption implementation with a hardcoded key will score the same as a properly managed one at Tiers 1–3. Code review remains necessary for correctness.

Novel implementations may be missed. AST queries and language packs are built around common implementation patterns. A project that implements encryption via a custom protocol rather than standard libraries may produce low-confidence results even if the implementation is sound. The calibration phase reduces but does not eliminate this gap.

Call graph analysis has boundaries. Static call graphs cannot resolve dynamic dispatch, reflection, runtime code generation, or cross-process communication. Features implemented via plugin systems, message queues, or microservice calls may appear disconnected even when they function correctly at runtime.

ARGUS triages; humans decide. Reports should be read as “here is the evidence we found and how deep it goes,” not “this software is compliant.” The tool’s value is in directing reviewer attention to the features with weakest evidence, not in replacing the review.

Failure Modes. False positives may arise from unused libraries present in dependency manifests, dead code paths that are syntactically reachable but never executed, or test fixtures that import features without exercising them. False negatives may occur when features are implemented via custom abstractions not covered by language packs, when behavior is determined dynamically at runtime, or when cross-service communication obscures local call graphs. Understanding these failure modes is essential for interpreting ARGUS output correctly.

Adversarial and Edge Cases. ARGUS does not attempt to detect intentionally misleading code, stub implementations designed to pass automated checks, or test spoofing (tests that assert trivially true conditions). A codebase constructed to deceive static analysis will likely produce high-confidence results that do not reflect genuine implementation quality. ARGUS assumes good-faith codebases; adversarial resilience is outside its design scope.

Scope Boundary. ARGUS is not a security audit tool, not a performance validator, and not a compliance certifier. It does not assess whether code is secure against known vulnerability classes, whether it meets performance benchmarks, or whether it satisfies regulatory standards. ARGUS identifies structural evidence that features appear to be implemented and reports the depth of that evidence. Compliance, security, and performance determinations remain the responsibility of qualified human reviewers using ARGUS output as one input among many.

Constraints & Assumptions

The agent performs static analysis only. It does not execute, compile, or runtime-test the target software. Evidence quality depends on the depth of analysis possible for each language (AST and call graph support varies).

Language support requires tree-sitter grammars and pre-built AST query patterns. Languages without a tree-sitter grammar are limited to Tier 1 evidence (file patterns, keywords, and dependency checks only).Supported languages with full Tier 1–4 capability include: Python, JavaScript/TypeScript, Go, Java, C#, Rust, and C/C++. All other languages receive Tier 1 analysis only. When a language lacks tree-sitter support, ARGUS outputs a warning flag in the report indicating degraded analysis depth. Expected degradation behavior is documented per tier for each unsupported language category.

All processing is local. No data leaves the machine unless LLM disambiguation mode is explicitly enabled by the operator. When LLM mode is enabled, only ambiguous code snippets are transmitted, never the full codebase.

All analyses are deterministic given identical inputs, configuration, and language pack versions, enabling reproducible results across environments. Configuration files, AST query patterns, and language pack versions are tracked to ensure that any result can be independently reproduced.

Benchmarking Plan

Future validation will benchmark ARGUS against manually reviewed codebases to quantify detection accuracy and identify systematic bias. Benchmarking will measure precision, recall, and

F1 across each evidence tier and each supported language. Results will be published alongside the tool to provide users with empirically grounded expectations for detection reliability. The benchmarking dataset will be versioned and made available for independent reproduction.

Offline Deployment Options

ARGUS comprises a Python review engine with tree-sitter bindings, a lightweight HTTP server, and a self-contained HTML dashboard. All components run at the edge with zero internet dependency.

However, the current implementation has external dependencies: PyYAML for configuration parsing and tree-sitter language grammars for AST analysis. Installing these requires pip, which requires internet access. This creates a gap in the offline deployment story: a machine that has never been connected to the internet cannot run ARGUS out of the box.

This section evaluates four options for eliminating that gap, each with different tradeoffs around complexity, user experience, and maintainability.

Option 1: Vendor Dependencies into the Project

RECOMMENDED

Copy the pure-Python PyYAML source files and pre-compiled tree-sitter grammar binaries directly into the ARGUS project folder. Python's import system finds them locally, eliminating the need for pip entirely. The only prerequisite is a Python 3.x installation on the target machine.

Option 2: Compile to Standalone Executable

STRONG

Use PyInstaller or cx_Freeze to bundle the Python runtime, all dependencies, tree-sitter grammar, and the HTML dashboard into a single executable. The end user double-clicks to run. No Python installation required.

Option 3: JSON-Only Configuration

VIABLE

Remove YAML support entirely and switch the configuration format to JSON, which Python handles natively. This eliminates the PyYAML dependency but does not address tree-sitter. It must be combined with Option 1 or 2 for full offline AST capability.

Option 4: Hybrid JSON + YAML Support

CONDITIONAL

Default to JSON for universal compatibility but also accept YAML configuration files if PyYAML is available on the host machine. The system auto-detects the format based on file extension and gracefully degrades to JSON-only when YAML support is absent.

Recommendation

Option 1 (Vendor Dependencies) is the recommended path. It preserves the best user experience: human-readable YAML configuration, full AST analysis capability, no install steps, no format migration. The only requirement is that the target machine has Python installed. The overhead is manageable: vendored Python files plus pre-compiled tree-sitter grammar for each target platform.

For environments where even Python cannot be guaranteed, Option 2 (standalone executable) provides a strong fallback, though it requires a one-time build on a connected machine.

Comparison Matrix