# **Multi-Language Security Analysis Strategy**

## **I. Introduction**

### **A. Purpose and Scope**

This report outlines a comprehensive strategy for performing automated security analysis across software projects utilizing multiple programming languages. The primary goal is to establish a robust, scalable, and efficient process for identifying security vulnerabilities, managing findings, and prioritizing remediation efforts within complex codebases. The scope encompasses repositories containing Scala, Node.js (JavaScript/TypeScript), Python, Java, and React (JSX/TSX), addressing the unique challenges posed by polyglot environments. This strategy details methodologies for language detection, tool selection, configuration, analysis execution, output processing, and validation, aiming to provide a practical framework for enhancing application security posture.

### **B. Target Languages and Challenges**

The target languages – Scala, Node.js, Python, Java, and React (JSX/TSX) – represent a common mix found in modern web applications and backend systems. Each language possesses distinct ecosystems, syntax, common vulnerability patterns, and associated security tooling. React, while technically a JavaScript library, introduces JSX/TSX syntax and specific security considerations (e.g., XSS via props, state management vulnerabilities 1).

Managing security analysis across these diverse languages presents several challenges:

1. **Accurate Language Identification:** Reliably detecting the specific language of each file, especially in mixed-language repositories or files containing embedded code (e.g., JavaScript in HTML/JSX), is crucial for selecting the appropriate analysis tools.3
2. **Tool Heterogeneity:** Different security tools (SAST, SCA, secret scanning) offer varying levels of support, accuracy, and rule coverage for each language and its frameworks.5 Selecting and orchestrating a suite of tools that provides adequate coverage across all target languages is complex.
3. **Framework Awareness:** Security analysis tools need to understand the specific frameworks used within each language (e.g., Akka/Play for Scala 11, Express/NestJS for Node.js 12, Django/Flask for Python 13, Spring/Jakarta EE for Java 14) to accurately identify vulnerabilities and reduce false positives.15
4. **Output Normalization:** Each tool often produces results in its own proprietary format. Normalizing these diverse outputs into a common format (like SARIF 19) is essential for consistent processing, aggregation, and reporting.22
5. **False Positive Management:** False positives are inherent in static analysis. Managing them effectively across multiple languages and tools requires consistent strategies for suppression, tuning, and feedback.23
6. **Deduplication:** Identical vulnerabilities may be reported multiple times by different tools or across different scans. Robust deduplication is needed to avoid redundant effort.27
7. **Scalability and Performance:** Scanning large, polyglot codebases efficiently requires optimizing scan times and resource consumption for each language and tool.28

This strategy aims to address these challenges systematically.

## **II. Language Detection**

Accurate identification of the programming languages present within a repository is the foundational step for applying the correct security analysis tools and rules. An effective language detection strategy must handle individual files, mixed-language files, and entire repositories reliably.

### **A. Methodology (File Extensions, Content Analysis, Heuristics)**

A multi-faceted approach is recommended for language detection:

1. **File Extension Matching:** This is the simplest and fastest method. A mapping between common file extensions and languages (e.g., .scala for Scala 30, .js/.mjs/.cjs for JavaScript 30, .py for Python 30, .java for Java 30, .jsx/.tsx for React 30) provides a quick initial classification. Tools like Linguist and enry maintain extensive lists of such mappings.30
2. **Content Analysis (Keywords, Shebangs, Modelines):** When file extensions are ambiguous or missing, analyzing file content becomes necessary.
   * **Shebangs:** Lines like #!/usr/bin/env python strongly indicate a Python script.31
   * **Keywords/Syntax:** Searching for language-specific keywords (e.g., import vs. require, def vs. function, class, specific operators) or syntax patterns can help differentiate languages.
   * **Modelines:** Vim or Emacs modelines within comments (e.g., /\* vim: set ft=cpp: \*/) can explicitly declare the language.31
   * **Framework/Library Identifiers:** The presence of framework-specific imports or configurations (e.g., import React from 'react', import { Component } from '@angular/core') can pinpoint language variants or specific technologies.
3. **Heuristics and Classifiers:** For more challenging cases or disambiguation (e.g., differentiating JavaScript from TypeScript, distinguishing domain-specific languages), more advanced techniques are employed:
   * **Heuristics:** Rule-based approaches encoded in tools like Linguist apply specific logic based on file structure or content patterns (e.g., \_\_init\_\_.py filename for Python 31). These are often refined based on large datasets like GitHub repositories.
   * **Statistical Classifiers:** Tools like Linguist use a Naive Bayes classifier trained on code samples as a fallback mechanism.33 More advanced ML models like GitHub's OctoLingua utilize features like token frequency and special character presence for higher accuracy, especially on files without extensions.33 Enry also employs a Bayesian classifier.32

### **B. Tooling (Linguist, enry)**

Several open-source tools specialize in language detection for code:

1. **GitHub Linguist:** The de facto standard used by GitHub itself. It employs a combination of strategies including file extensions, filenames, shebangs, modelines, XML headers, heuristics, and a Naive Bayes classifier.31 While widely used, its performance can degrade on files without extensions, and its reliance on Ruby might be a consideration for some toolchains.33 It maintains a comprehensive languages.yml file detailing known languages, extensions, and interpreters.30
2. **go-enry:** A Go port of Linguist, designed for improved performance (claiming up to 2x speed increase).32 It uses a similar sequence of strategies (filename, extension, shebang, modeline, content heuristics, classifier) but leverages Go's performance characteristics.32 It offers a CLI and Go library, potentially integrating more easily into Go-based tooling.32 Enry also notes some divergences from Linguist due to limitations in Go's regex engine and differences in the classifier implementation.32

The choice between Linguist and enry may depend on performance requirements and the existing technology stack. For consistency with GitHub's language statistics, Linguist might be preferred, while enry could offer performance benefits.

### **C. Handling Mixed-Language Repositories (File-level detection, Dominant language)**

Polyglot repositories require a language detection strategy that operates at the file level.

1. **File-Level Granularity:** The detection process must be applied to each file individually using the methodologies described above (extension, content, heuristics). This ensures that, for example, Python scripts, Java backend code, and React frontend code within the same repository are correctly identified.
2. **Mixed-Language Files:** Files containing multiple languages (e.g., JavaScript embedded in HTML/JSX, SQL within Java strings) pose a challenge. Most language detection tools identify a single primary language for a file. SAST/SCA tools themselves often have capabilities to parse and analyze embedded languages within a primary file type (e.g., scanning JavaScript within an HTML file). The language detection step should primarily focus on identifying the main language to route the file to the appropriate *primary* scanner.
3. **Repository-Level Summary:** While file-level detection is crucial for tool selection, generating a repository-level summary (e.g., percentage breakdown of languages) can be useful for reporting and understanding the overall composition. Tools like Linguist provide this summary, often identifying a "dominant" language but still tracking all detected languages.

The strategy should rely on file-level detection to trigger the appropriate language-specific security scanners within the CI/CD pipeline.

## **III. Tool Selection and Configuration**

Selecting the right security analysis tools and configuring them appropriately for each language is critical for achieving effective and efficient vulnerability detection.

### **A. Criteria (Language Support, Rule Coverage, Framework Awareness, Output Format, Integration, Performance, Cost)**

The selection process for SAST, SCA, and secret scanning tools should be guided by the following criteria:

1. **Language Support:** Verify explicit and robust support for the target versions of Scala, Node.js (JavaScript/TypeScript), Python, Java, and React (JSX/TSX).10 Check for nuances like support for specific ECMAScript versions or Java bytecode versions.35
2. **Rule Coverage & Quality:**
   * **Breadth:** Assess the number and types of vulnerabilities the tool can detect (e.g., OWASP Top 10 36, SANS Top 25, CWEs 38).
   * **Depth:** Evaluate the quality and precision of the rules. Does the tool support taint analysis (tracking data flow from source to sink) for injection vulnerabilities? 16
   * **Language-Specificity:** Ensure rules cover common pitfalls and vulnerability patterns specific to each language (e.g., Scala Future handling 41, Java reflection issues 2, Node.js prototype pollution 43, Python template injection 45, React XSS/state issues 1).
   * **False Positive Rate:** Consider the tool's historical accuracy and reported false positive rates. Tools with high false positive rates can lead to alert fatigue and wasted effort.23 Look for user reviews or benchmark comparisons.48
3. **Framework Awareness:** Crucial for accuracy, especially with SAST. The tool should understand common frameworks used with the target languages (e.g., Akka/Play for Scala 11, Express/NestJS for Node.js 12, Django/Flask for Python 13, Spring/Jakarta EE for Java 14, React itself 9) to correctly trace data flows and identify framework-specific vulnerabilities.15 Lack of framework awareness often leads to false positives or negatives.
4. **Output Format:** Standardized output is essential for aggregation and normalization. Prioritize tools that support SARIF 2.1.0 19 or provide well-documented JSON/XML formats that can be easily transformed into SARIF. Check for support for CycloneDX VEX for SCA tools.60
5. **Integration Capabilities:**
   * **CI/CD:** Assess ease of integration into existing CI/CD pipelines (e.g., GitHub Actions 63, GitLab CI, Jenkins 38). Look for official plugins or well-documented CLI options.
   * **IDE:** IDE plugins (e.g., SonarLint 65, Semgrep VSCode extension 28) provide immediate feedback to developers ("shift-left").7
   * **Vulnerability Management Platforms:** Check for APIs or native integrations with platforms like DefectDojo 66 or Faraday.68
6. **Performance:** Evaluate scan speed and resource consumption (CPU, memory).28 Consider support for incremental scanning 77 and caching.78 Performance can vary significantly by language and tool architecture.
7. **Customization:** The ability to create custom rules or detectors is vital for addressing organization-specific coding standards or unique vulnerability patterns.79 Assess the ease of rule development and management.
8. **Support and Maintenance:** Consider the tool's maintenance frequency, community support (for open source), or vendor support responsiveness (for commercial tools). Check for active development and regular updates to rules and the scanning engine.33
9. **Cost:** Evaluate licensing models (open source, free tier, per user, per application, per scan) and total cost of ownership, including infrastructure and maintenance.5

### **B. Language-Specific Tool Recommendations (SAST, SCA, Secrets)**

Based on the criteria above and available research, the following tools are recommended candidates for each language. Note that a layered approach, potentially using multiple tools per language category (SAST, SCA, Secrets), often provides the best coverage.

1. **Scala:**
   * **SAST:**
     + *SonarQube:* Offers Scala support 51, including importing Scalastyle/Scapegoat reports.51 Rule coverage details need verification via their rules catalog.37 Framework support for Akka/Play is listed in some documentation 11 but needs confirmation for depth.
     + *Semgrep:* Lists Scala as Generally Available (GA).88 Known for strong custom rule capabilities.79 Framework support needs verification.
     + *Checkmarx:* Supports Scala, including Akka and Finagle/Finatra frameworks.11 Commercial tool with likely extensive rule sets.77
     + *Find Security Bugs (SpotBugs Plugin):* Supports Scala 10 and focuses on security bug patterns.38
   * **SCA:**
     + *OWASP Dependency-Check:* Has an SBT plugin 89, indicating Scala support.
     + *Checkmarx SCA:* Supports SBT.90
     + *Snyk:* Supports Scala via SBT and Maven.91
     + *Veracode SCA:* Supports Scala.92
     + *Mend SCA:* Supports Scala.93
   * **Secrets:** Gitleaks 85, TruffleHog 86 (language agnostic).
2. **Node.js (JavaScript/TypeScript):**
   * **SAST:**
     + *SonarQube:* Strong support for JS/TS, including React JSX and Vue.js.81 Offers custom rules.81 Framework support for Express/NestJS needs verification via rules catalog.37
     + *Semgrep:* GA support for JS/TS.94 Specific framework support for Express, NestJS, Hapi, Koa, React, Angular.56 Strong custom rule capabilities.43
     + *ESLint:* Highly configurable linter for JS/TS.96 Can enforce security rules via plugins (e.g., eslint-plugin-security, eslint-plugin-react, eslint-plugin-react-hooks).97 Supports JSX.99 Custom rules are well-supported.84
     + *Checkmarx:* Supports JavaScript and TypeScript, including Node.js, ExpressJS, React.13
     + *Veracode:* Supports JavaScript/AJAX, TypeScript, Node.js.92
     + *Snyk Code:* Supports JavaScript/TypeScript.102
     + *Mend SAST:* Supports JavaScript/TypeScript.103
   * **SCA:**
     + *OWASP Dependency-Check:* Has an npm module.104
     + *Snyk:* Strong Node.js (npm, Yarn) support.91
     + *Checkmarx SCA:* Supports npm, Yarn, pnpm.107
     + *Veracode SCA:* Supports npm, Yarn.92
     + *Mend SCA:* Supports npm, Yarn.93
     + npm audit / yarn audit: Built-in package manager capabilities.
   * **Secrets:** Gitleaks 85, TruffleHog.86
3. **Python:**
   * **SAST:**
     + *SonarQube:* Supports Python 3.x, Flask, Django.53 Custom rules supported.82
     + *Semgrep:* GA support for Python.108 Framework support for Django, Flask.108 Strong custom rule capabilities.45
     + *Bandit:* Open-source tool specifically for finding common security issues in Python code.7 Supports custom plugins.111 Aware of Flask debug mode.112
     + *Checkmarx:* Supports Python, including Django and Flask frameworks.13
     + *Veracode:* Supports Python.92
     + *Snyk Code:* Supports Python.102
     + *Mend SAST:* Supports Python.103
   * **SCA:**
     + *OWASP Dependency-Check:* Supports Python analysis.113
     + *Snyk:* Supports Pip, Poetry, Pipenv.91
     + *Checkmarx SCA:* Supports Pip, Poetry, Setuptools.107
     + *Veracode SCA:* Supports Pip.92
     + *Mend SCA:* Supports Pip, Conda.93
     + *pip-audit:* Tool for auditing Python environments.
   * **Secrets:** Gitleaks 85, TruffleHog.86
4. **Java:**
   * **SAST:**
     + *SonarQube:* Strong Java support, including JSP.83 Supports Spring framework.54 Jakarta EE support needs verification.65 Custom rules supported.83 Taint analysis capabilities.40
     + *Semgrep:* GA support for Java.116 Framework support for Spring, Java Servlets.116 Strong custom rule capabilities.117
     + *SpotBugs / Find Security Bugs:* Bytecode analysis for Java.76 FindSecBugs plugin adds security-specific rules.38 Custom detectors can be written.119
     + *Checkmarx:* Supports Java, including Spring.13 Jakarta EE support needs verification.
     + *Veracode:* Supports Java.92 Spring/Jakarta EE support needs verification.121
     + *Snyk Code:* Supports Java.102 Spring support likely.
     + *Mend SAST:* Supports Java.103
   * **SCA:**
     + *OWASP Dependency-Check:* Strong Java support via CLI, Maven, Gradle plugins.122
     + *Snyk:* Supports Maven, Gradle.91
     + *Checkmarx SCA:* Supports Maven, Gradle, Ivy, SBT.124
     + *Veracode SCA:* Supports Maven, Gradle.92
     + *Mend SCA:* Supports Maven, Gradle, Ant.93
   * **Secrets:** Gitleaks 85, TruffleHog.86
5. **React (JSX/TSX):** (Primarily covered by JavaScript/TypeScript tools)
   * **SAST:**
     + *SonarQube:* Supports React JSX.81 Recently added specific React rules.125
     + *Semgrep:* GA support for JSX/TSX.94 Framework support for React, Next.js, Angular.56 Custom rules are effective.126
     + *ESLint:* Core tool for JS/TS linting. Requires eslint-plugin-react and eslint-plugin-react-hooks for React-specific rules (e.g., checking dangerouslySetInnerHTML usage, hook rules).97 Custom rules can target JSX AST nodes.84
     + *Checkmarx:* Supports JavaScript/TypeScript, including React.13
     + *Veracode:* Supports JavaScript/TypeScript.92 React support likely.
     + *Snyk Code:* Supports JavaScript/TypeScript.102 React support likely.
     + *Mend SAST:* Supports JavaScript/TypeScript.103 React support likely.
   * **SCA:** (Covered by Node.js SCA tools - npm/Yarn)
     + *OWASP Dependency-Check:* Supports Node.js dependencies.104
     + *Snyk:* Supports npm, Yarn.91
     + *Checkmarx SCA:* Supports npm, Yarn, pnpm.107
     + *Veracode SCA:* Supports npm, Yarn.92
     + *Mend SCA:* Supports npm, Yarn.93
   * **Secrets:** Gitleaks 85, TruffleHog.86

*Note: Tool capabilities, especially for commercial products and specific framework/rule coverage, should be verified directly with vendor documentation as sources may be outdated or inaccessible.*

### **C. Framework and Library Detection**

Effective SAST, particularly taint analysis, relies heavily on understanding the frameworks and libraries used in the application.

1. **Tool Capabilities:** Many modern SAST tools have built-in awareness of common frameworks.15
   * *SonarQube:* Has known support for frameworks like Spring 54 and aims to understand security controls within popular frameworks.16 Its support for others like Akka, Play, Express, NestJS, Django, Flask, Jakarta EE needs verification through its rule documentation.37
   * *Semgrep:* Explicitly lists support for frameworks like Express, NestJS, React, Angular, Django, Flask, Spring, Java Servlets, Go net/http.56 Its Pro Engine enhances cross-file and cross-function analysis, crucial for tracking data flow through framework layers.131
   * *Checkmarx:* Supports a wide range of frameworks across languages, including Akka, Play, Express, NestJS, Django, Flask, Spring.11
   * *Bandit:* Has specific checks for Flask debug mode 112 and some Django patterns.111 Framework awareness is less extensive than broader SAST tools but can be extended via plugins.111
   * *FindSecBugs:* Includes checks for frameworks like Spring-MVC, Struts, Tapestry.38
2. **Custom Rules/Configuration:** When built-in framework awareness is insufficient or for custom/internal frameworks, custom rules or configurations are necessary.
   * *Taint Analysis Configuration:* Tools like SonarQube 132 and Semgrep 39 allow defining custom sources (where untrusted data enters, e.g., specific request parameters in a framework), sinks (where data causes harm, e.g., database query execution), and sanitizers/validators (functions that make data safe).
   * *Pattern Matching:* Tools like Semgrep allow writing patterns that match framework-specific function calls or configurations.95
   * *AST/Bytecode Analysis:* Tools like ESLint 97, Bandit 111, and SpotBugs 120 allow writing custom rules/plugins that analyze the Abstract Syntax Tree (AST) or bytecode to understand framework usage.

Accurate framework detection and analysis are vital for reducing false positives (e.g., not flagging a sink if framework sanitization is used) and finding true positives (e.g., tracing tainted data through framework abstractions).

### **D. Custom Rule Configuration**

Custom rules are essential for tailoring security analysis to specific organizational needs, coding standards, and application contexts beyond the coverage of generic rulesets.

1. **Necessity and Use Cases:**
   * *Enforcing Secure Coding Standards:* Mandate specific security practices not covered by default rules (e.g., use of specific internal crypto libraries, banning deprecated internal APIs).7
   * *Framework-Specific Logic:* Detect vulnerabilities unique to custom frameworks or specific configurations of standard frameworks.79
   * *Business Logic Flaws:* Identify potential security issues tied to specific application business logic.
   * *Reducing False Positives:* Refine existing rules or create more specific versions to eliminate known false positives in the organization's context.98
   * *Targeting Newly Discovered Vulnerabilities:* Quickly create rules to scan for variants of newly disclosed vulnerabilities before official rules are available.133
2. **Language-Specific Examples:**
   * **Scala Futures:** Detect potential issues like blocking operations outside a blocking {} context or improper exception handling within Future callbacks.41 *Syntax Guidance (Semgrep - Conceptual):* Use patterns to find Future blocks and pattern-not-inside to check for blocking {}, or analyze method calls within onComplete, map, flatMap. 79
   * **Node.js Prototype Pollution:** Detect patterns where user-controlled input might modify Object.prototype.43 *Syntax Guidance (Semgrep):* Use taint mode (mode: taint) with sources like request inputs (req.query, req.body) and sinks involving object merging or property assignment using potentially controlled keys (e.g., target[key] = source[key], \_.merge). 39
   * **Python Template Injection (Django/Flask):** Detect unsafe template rendering with user-controlled input.45 *Syntax Guidance (Semgrep):* Use taint mode, marking user input sources and sinks like Django's mark\_safe or direct string formatting used in template context creation. Check for disabled autoescaping.13439 *Syntax Guidance (Bandit):* Write plugins checking AST nodes for specific unsafe template rendering patterns.111
   * **Java Insecure Reflection:** Detect reflection usage that could be manipulated by untrusted input (e.g., loading classes or invoking methods based on user strings).42 *Syntax Guidance (SonarQube):* Implement custom checks visiting relevant AST nodes (e.g., MethodInvocationTree for Class.forName, Method.invoke) and analyzing arguments for taint.80 *Syntax Guidance (SpotBugs/FindSecBugs):* Write detectors analyzing bytecode for calls to reflection APIs (java.lang.reflect.\*) and tracing data flow to identify user control.119
   * **React XSS via Props/dangerouslySetInnerHTML:** Detect unsanitized data passed to dangerouslySetInnerHTML or potentially risky props.2 *Syntax Guidance (ESLint):* Create rules visiting JSXAttribute nodes. Check if name.name is dangerouslySetInnerHTML. If so, trace the source of the \_\_html value to ensure it passes through a known sanitizer (e.g., DOMPurify.sanitize).128 Analyze props passed to components known to render HTML. 84
3. **Rule Management:**
   * **Version Control:** Store custom rules in a dedicated version-controlled repository (e.g., Git) for tracking changes, collaboration, and history.
   * **Testing:** Implement test cases (both positive and negative examples) for each custom rule to ensure accuracy and prevent regressions. Tools like Semgrep 95 and ESLint 84 have built-in testing frameworks.
   * **Documentation:** Document the purpose, logic, potential false positives, and remediation guidance for each custom rule.
   * **Review Process:** Establish a peer review process for new or modified custom rules to maintain quality and consistency.136
   * **Organization:** Organize rules logically (e.g., by language, vulnerability type, framework) for easier management and application. Platforms like Semgrep AppSec Platform allow policy management.137

## **IV. Analysis Execution and Optimization**

Integrating and optimizing security scans within the development workflow is crucial for timely feedback and efficient resource utilization.

### **A. Orchestration in CI/CD Pipelines**

Automating security scans within CI/CD pipelines ensures consistent checks and early vulnerability detection.

1. **Integration Points:** Trigger scans at appropriate stages of the SDLC 64:
   * *Pre-Commit (Optional):* Lightweight checks (e.g., secret scanning, basic linting) can run locally using pre-commit hooks.139
   * *Pull Request/Merge Request:* Scan changed code/files to provide feedback before merging (SAST, Secrets). This is a critical point for "shifting left".
   * *Build Stage:* Perform more comprehensive scans after compilation (SAST on bytecode, SCA on built artifacts).
   * *Post-Deployment (Test/Staging):* Run DAST scans against running applications (outside the scope of this multi-language *static* analysis strategy but part of a holistic approach).
2. **Toolchain Orchestration:** Efficiently manage the execution of multiple tools across different languages.
   * *Language Detection First:* Run language detection (e.g., enry 32) early in the pipeline to determine which scanners need to be invoked for the changed files or the entire repository.
   * *Conditional Execution:* Trigger specific scanners only if relevant languages are detected or files with specific extensions are modified.
   * *Parallelization:* Run scanners for different languages or independent parts of the codebase in parallel (-j flag in Semgrep) where possible to reduce overall pipeline time. Consider runner resource limits.
   * *Caching:* Implement caching mechanisms for tool installations, dependency downloads (SCA), and potentially analysis results (e.g., ESLint --cache 78, SonarQube analysis cache) to speed up subsequent runs.
3. **Containerization Strategy:** Using Docker containers for security tools provides environment consistency and simplifies dependency management.139
   * **Official/Recommended Images:** Prioritize using official or well-maintained Docker images for each tool:
     + *Semgrep:* semgrep/semgrep 147 (Alpine-based, includes Python 3.11, bash, jq, curl). Note non-root variants like latest-nonroot.149
     + *Bandit:* ghcr.io/pycqa/bandit/bandit 143 (multi-arch, signed). Avoid potentially outdated community images like secfigo/bandit 153 or opensorcery/bandit.154
     + *ESLint:* No single official image, but community images exist (e.g., pipelinecomponents/eslint 155). Often run via npx within a Node.js container.78 Base on official Node.js images following best practices.157
     + *SpotBugs/FindSecBugs:* No official SpotBugs image found.158 FindSecBugs community images exist (e.g., shibme/findsecbugs-docker 162) but may be outdated. Often run via Maven/Gradle plugins within a JDK container.118
     + *SonarScanner CLI:* sonarsource/sonar-scanner-cli.165 Requires Java runtime.
     + *OWASP Dependency-Check:* Run via CLI (requires Java 105) or build tool plugins (Maven 123, Gradle 169, SBT 170) within appropriate container (JDK, Node.js).
     + *Syft:* anchore/syft.171 Chainguard offers a minimal Wolfi-based image cgr.dev/chainguard/syft.175
     + *Grype:* anchore/grype.63 Chainguard offers minimal Wolfi-based images cgr.dev/chainguard/grype 183 and grype-fips.184
     + *LicenseFinder:* licensefinder/license\_finder.139
     + *Gitleaks:* Official image available (details not in snippets, assume gitleaks/gitleaks).
     + *TruffleHog:* Official image available (details not in snippets, assume trufflesecurity/trufflehog).
   * **Volume Mapping:** Mount necessary volumes into the containers:
     + *Source Code:* Mount the repository checkout directory (e.g., -v $(pwd):/src 153, -v "${YOUR\_REPO}:/usr/src" 166). Ensure the working directory inside the container matches where the tool expects code (e.g., /src for Semgrep 149, /code for Bandit 153).
     + *Configuration Files:* Mount tool-specific configuration files (e.g., .semgrepignore, .bandit.yml 187, .grype.yaml 188, custom rules) into expected locations or specify paths via CLI arguments.176
     + *Cache Directories:* Persist tool caches (e.g., Grype DB 176, ESLint cache 78, SonarScanner cache 166) using named Docker volumes or host mounts to speed up subsequent scans. Default Grype cache path pattern: $XDG\_CACHE\_HOME/grype/db/<SCHEMA-VERSION>/.176 Default SonarScanner cache: /opt/sonar-scanner/.sonar/cache.166
     + *Output Directories:* Mount a host directory or use shared volumes to collect report files (e.g., SARIF, JSON) generated by the tools.153
   * **User Permissions:** Be mindful of the user running inside the container. Some official images run as non-root (e.g., semgrep/semgrep:latest-nonroot uses user semgrep 149, sonarsource/sonar-scanner-cli uses user scanner-cli UID 1000 165). Ensure mounted volumes have appropriate permissions for the container user to read code and write caches/reports. Running as root should be avoided where possible.191 Anchore Grype/Syft images based on distroless/scratch likely run as root by default, but check specific image layers.173
   * **Resource Allocation:** Define appropriate CPU and memory limits for scanner containers based on tool recommendations and observed usage.72 Insufficient memory is a common cause of scan failures, especially for complex SAST tools.29 Semgrep recommends --max-memory flag.194

### **B. Handling Mixed-Language Repositories**

Orchestration needs specific logic for polyglot repositories:

1. **Scan Triggering:** Trigger scans based on file changes detected by the CI system. Use the language detection results for the changed files to invoke only the relevant scanners. For full repository scans (e.g., scheduled nightly builds), run language detection on the entire repo and trigger all necessary scanners.
2. **Scope Definition:** Define the scan scope appropriately. For a full scan, provide the root directory to each relevant scanner. For PR scans, configure tools (if supported) to only analyze changed files or code regions to optimize speed. Tools like Semgrep CI (semgrep ci) are designed for diff-aware scanning.195

### **C. Analysis Speed Optimization**

Optimizing scan speed is crucial for maintaining fast feedback loops in CI/CD.

1. **Language-Specific Considerations:**
   * *Compiled (Java, Scala):* SAST tools may operate on source code or bytecode. Bytecode analysis (e.g., SpotBugs 76) requires a compilation step first. Source code analysis might be faster if compilation is slow, but bytecode analysis can sometimes find different types of issues.
   * *Interpreted (Python, Node.js):* Scans typically operate directly on source code. Performance can be affected by the size and complexity of the codebase and its dependencies.
   * *Transpiled (TypeScript, JSX):* Tools might analyze the original source (TSX/JSX) or the transpiled JavaScript. Analyzing the source is generally preferred for accurate location mapping and rule application. Ensure tools correctly handle the specific syntax (e.g., ESLint with appropriate parser/plugins 99).
   * *AST Size/Complexity:* Languages with complex syntax or large generated ASTs might require more memory and time for analysis.
2. **Tool Configuration:**
   * *Incremental Scans:* Leverage incremental scanning features where available (e.g., Checkmarx SAST incremental=true 77) to only analyze changes since the last scan.
   * *Rule Subsetting:* Run only relevant or high-priority rulesets during PR checks, reserving full ruleset scans for nightly or release builds. Configure tools to exclude specific rules or checks known to be slow or irrelevant (-s/--skips in Bandit 196, Semgrep configuration 26).
   * *Caching:* Utilize tool-specific caching mechanisms (e.g., ESLint --cache 78, Grype DB caching 176, SonarQube analysis cache) to avoid redundant work. Ensure cache volumes are correctly mounted and persisted in CI.
   * *Timeouts:* Configure timeouts (--timeout in Semgrep 28) to prevent runaway scans on complex files, potentially skipping problematic files after a threshold (--timeout-threshold 28).
   * *Parallelism:* Adjust the number of parallel jobs (-j in Semgrep 28) based on available CI runner resources.
3. **Resource Allocation:**
   * *CPU/Memory:* Allocate sufficient CPU and memory to scanner containers/jobs. Monitor resource usage during initial runs to establish baseline requirements.72 Memory exhaustion is a common failure point for SAST.194 Tools like Semgrep have memory limiting flags (--max-memory).194 Performance can degrade significantly if resources are constrained.70

## **V. Output Processing and Management**

Raw output from diverse security tools needs to be processed, normalized, enriched, and managed effectively to enable efficient triage and remediation.

### **A. Output Normalization**

Consolidating findings from multiple tools requires a standardized format.

1. **Need for Standardization:** Different tools produce outputs with varying schemas, severity levels, and levels of detail. A common format simplifies aggregation, deduplication, reporting, and integration with vulnerability management platforms.22
2. **SARIF as the Standard Format:** The Static Analysis Results Interchange Format (SARIF) version 2.1.0 is the industry standard designed for this purpose.21 It provides a structured JSON schema for representing static analysis findings.
   * **Structure:** A SARIF log file (sarifLog object) contains one or more runs. Each run represents an execution of an analysis tool (tool.driver) and contains an array of results.19
   * **Key Fields in result:**
     + ruleId: Identifies the specific rule violated (e.g., "SQLInjectionRisk", "CWE-89").19 Crucial for tracking and deduplication.59
     + level: Indicates severity ("error", "warning", "note", "none").19
     + message: Contains a human-readable description (text and/or markdown) of the finding.19
     + locations: An array specifying where the finding occurred. Crucially includes physicalLocation.19
     + physicalLocation: Contains artifactLocation (with uri for the file path) and region (with startLine, startColumn, endLine, endColumn for precise code location).19 Consistent file path representation is vital.59
     + partialFingerprints/fingerprints: Dictionaries used for uniquely identifying and deduplicating findings across runs.22 See section V.D.
     + codeFlows: Represents execution paths relevant to the finding.19
     + relatedLocations: Points to other relevant code locations.19
   * **Tool/Rule Metadata:** The run.tool.driver object contains metadata about the scanner and its rules (rules array of reportingDescriptor objects), including rule descriptions (shortDescription, fullDescription), help URIs (helpUri), and properties like tags and severity (properties.tags, defaultConfiguration.level, properties.security-severity).19
3. **VEX for Exploitability:** For SCA findings, Vulnerability Exploitability eXchange (VEX), often using the CycloneDX format, communicates whether a vulnerability in a dependency is actually exploitable in the specific product context.61 This aids prioritization by filtering out vulnerabilities in unused code paths.
   * **CycloneDX VEX analysis Object:** Contains fields to describe the vulnerability's status 60:
     + state: The assessment status (e.g., not\_affected, affected, fixed, under\_investigation).60
     + justification: The rationale if state is not\_affected (e.g., code\_not\_present, code\_not\_reachable, requires\_configuration, protected\_by\_mitigating\_control).60
     + response: Actions taken or planned (e.g., can\_not\_fix, will\_not\_fix, update, rollback).60
     + detail: A detailed explanation of the analysis and justification.60
4. **Tool-Specific SARIF Support:** Assess SARIF output capabilities of selected tools.
   * *Semgrep:* Native SARIF output (--sarif, --sarif-output).201
   * *Snyk:* Supports SARIF output for snyk code test (SAST) and snyk test (SCA) via --sarif and --sarif-file-output flags.203
   * *Checkmarx:* Provides SARIF generation capabilities, potentially via CLI or specific modules/APIs.209
   * *Veracode:* Offers SARIF export, potentially via API or specific actions/tools like veracode-pipeline-scan-results-to-sarif.214
   * *Bandit:* Supports SARIF output format (-f sarif).187
   * *SpotBugs/FindSecBugs:* SARIF support might be available via IDE plugins 218 or specific actions like spotbugs-findsecbugs-action 219, but core tool or Maven/Gradle plugin support needs verification.220
   * *OWASP Dependency-Check:* Supports SARIF format via CLI (--format SARIF) and Maven plugin (<format>SARIF</format>).222 Gradle plugin support needs verification.169 SBT plugin support needs verification.170
   * *Gitleaks:* Supports SARIF output (-f sarif).85
   * *TruffleHog:* SARIF support not explicitly confirmed; JSON output available.86 Check issues/docs.225
   * *Other Tools (JFrog Xray, Black Duck, etc.):* Check documentation for SARIF export capabilities.227
   * *Platforms (GitHub, GitLab):* GitHub Actions 59 and GitLab CI 231 natively support ingesting SARIF reports for their security dashboards.

If a tool lacks native SARIF support, develop a custom transformer to convert its native JSON/XML output to the SARIF 2.1.0 schema.

### **B. Severity Normalization**

Aggregating findings requires mapping diverse tool-specific severity ratings (e.g., "Critical", "High", "P1", numeric scores) to a consistent internal scale.

1. **Challenges with Diverse Scales:** Tools use different labels and underlying methodologies (CVSS versions, custom scoring) making direct comparison difficult.232 A "High" in one tool might not equate to a "High" in another.
2. **Using CVSS as a Baseline:** The Common Vulnerability Scoring System (CVSS) is the industry standard for vulnerability severity.232
   * *Mapping:* Map tool severities to CVSS v3.1 (or latest) qualitative ratings (Low: 0.1-3.9, Medium: 4.0-6.9, High: 7.0-8.9, Critical: 9.0-10.0) where possible.232 Some tools provide CVSS scores directly.
   * *Limitations:* CVSS base scores are static and lack organizational context (e.g., asset criticality, internet exposure).232 A high CVSS score might represent low actual risk in a specific environment, and vice-versa. Relying solely on CVSS for prioritization is insufficient.232
3. **Defining an Internal Severity Scale:** Establish a clear, documented internal severity scale (e.g., Critical, High, Medium, Low, Informational).
   * *Mapping Rules:* Define explicit rules for mapping each source tool's severity levels (and CVSS scores, if available) to the internal scale. Example: Map Tool X 'Severe' and CVSS >= 9.0 to Internal 'Critical'; Map Tool Y 'P1' to Internal 'High'.
   * *Consistency:* Ensure the mapping logic is applied consistently during the normalization process.
   * *Review:* Periodically review and adjust the internal scale and mapping rules based on organizational risk appetite and observed threat landscape.

This normalized internal severity becomes a key input for prioritization, but should be combined with contextual enrichment (see Section VII).

### **C. False Positive Management**

Effectively managing false positives (FPs) is crucial to maintain developer trust and focus remediation efforts on genuine threats.23

1. **Language-Specific Patterns:** Be aware of common FP patterns for each language and tool combination.
   * *Scala:* Complex type system or functional constructs might confuse some SAST tools. Frameworks like Akka/Play can obscure data flow if not properly understood by the tool.87
   * *Node.js (JS/TS):* Dynamic typing, complex callback patterns, and prototype manipulation can lead to FPs in taint analysis.98 Framework abstractions (Express middleware, NestJS decorators) require tool awareness.81
   * *Python:* Dynamic typing and metaprogramming can challenge SAST. Frameworks like Django/Flask have specific security mechanisms (e.g., autoescaping) that tools need to recognize.53 Bandit, being Python-specific, might have fewer FPs for certain Python patterns but may need tuning.24
   * *Java:* Reflection 42, complex dependency injection (Spring), and bytecode analysis (SpotBugs 238) can be sources of FPs if analysis is imprecise.
   * *React (JSX/TSX):* Virtual DOM manipulation and state management patterns can be misinterpreted. Correctly identifying unsanitized data passed to dangerouslySetInnerHTML requires careful analysis.127 ESLint rules often require tuning for specific React patterns.127
2. **Reduction Techniques:** Employ a combination of techniques:
   * *Rule Tuning:* Adjust or disable overly noisy rules within the tool's configuration or the vulnerability management platform.25 Some tools allow fine-tuning rule parameters.196
   * *Contextual Analysis:* Use contextual information (see Section VII) to automatically deprioritize or filter findings unlikely to be exploitable (e.g., vulnerability in code not reachable from the internet).23
   * *Suppressions/Ignores:*
     + *In-code Annotations:* Use comments like # nosec (Bandit 24), //NOSONAR 25, or gitleaks:allow 85 to suppress specific findings on a line, ideally with justification.
     + *Configuration Files:* Define ignore rules based on vulnerability ID, file path, code patterns, etc., in tool configuration files (e.g., .semgrepignore 189, Bandit config 196, Grype config 176).
     + *Baseline Reports:* Use baseline reports (Bandit -b 241, Gitleaks --baseline-path 85) to ignore pre-existing, accepted findings in subsequent scans.
     + *Vulnerability Management Platform:* Mark findings as False Positive or Acceptable Risk within platforms like DefectDojo 244 or Faraday.68
3. **Feedback Loops and Confidence Scoring:** Implement a systematic process to learn from manual triage:
   * *Manual Triage Workflow:* Define clear procedures for analysts/developers to review findings and assign a status (True Positive, False Positive, Acceptable Risk, etc.).240 Capture the justification for FP/Accepted Risk decisions.
   * *Feedback Integration:*
     + *Periodic Review:* Regularly analyze FP trends for specific rules or tools. High FP rates for a rule may warrant tuning, replacement, or disabling.26
     + *Confidence Scoring:* Develop a confidence score (e.g., 0-100) for findings.250 Factors influencing the score can include:
       - *Rule Precision:* Historical True Positive vs. False Positive rate for the specific rule.47
       - *Tool Reputation/Reliability:* General confidence in the tool's accuracy for the specific language/check type.
       - *Clarity of Evidence:* How clearly the tool demonstrates the vulnerability (e.g., detailed data flow vs. simple pattern match).
       - *Contextual Factors:* Incorporate context like exploitability data (EPSS/KEV) or reachability analysis.240
       - *ML/AI Assistance:* Leverage ML models trained on historical triage data to predict the likelihood of a finding being a true positive.23 Tools like CrowdStrike Charlotte AI 245 and AquilaX 246 offer AI-powered triage.
     + *Score Updates:* Use the manual triage feedback (FP/TP markings) to continuously update the historical precision data and refine the confidence scores for rules/tools.
   * *Prioritization Based on Confidence:* Use confidence scores alongside severity and business context to prioritize triage efforts, focusing on high-confidence, high-severity findings first.250 Findings below a certain confidence threshold might be automatically deprioritized or flagged for quicker review.

### **D. Deduplication Strategies**

Accurately identifying unique vulnerabilities across different tools and scans is essential to avoid redundant triage and remediation efforts.

1. **Need for Robust Deduplication:** Simple line-based matching is fragile and breaks easily with code refactoring. Tools may report the same underlying vulnerability (e.g., a specific CVE in a library) at different locations or with slightly different descriptions.22 Effective deduplication requires identifying logically identical findings.
2. **Fingerprinting Techniques:** Generate a unique, stable identifier (fingerprint) for each finding based on its core characteristics.260
   * **Algorithms:**
     + *Cryptographic Hashes:* Use algorithms like SHA-256 (preferred) or SHA-1 (less secure but common) to hash relevant finding attributes.263 MD5 is generally discouraged due to collision weaknesses but might be seen in older systems.264 Faster, non-cryptographic hashes might be suitable if collision resistance requirements are lower.264
     + *Locality-Sensitive Hashing (LSH):* Techniques like MinHash can be used for approximate matching, grouping similar findings even if not identical.266 Useful for clustering text descriptions.
     + *Perceptual Hashing:* Primarily for multimedia, but the concept of hashing based on features rather than exact content can be relevant.263
   * **Attributes to Include in Fingerprint:** A robust fingerprint combines multiple attributes 267:
     + *Vulnerability Type:* Rule ID (tool-specific), CWE ID (Common Weakness Enumeration).
     + *Code Location:* Normalized file path (relative to repository root), line number(s) or range, potentially function/method name or signature (especially for AST-based fingerprinting).
     + *Code Snippet Hash:* Hash of the specific lines of code identified in the finding's location (region). This provides resilience against changes elsewhere in the file.
     + *Dependency Information (for SCA):* Package name, version, ecosystem (e.g., npm, Maven).
     + *(Optional) Contextual Elements:* Potentially include elements of the data flow path (source/sink information) for taint analysis findings.
   * **SARIF Fingerprints:** The SARIF standard includes dedicated fields for deduplication 22:
     + fingerprints: A dictionary of stable fingerprints generated according to different algorithms (key is algorithm name, value is fingerprint). Often populated by the consuming system (e.g., GitHub).
     + partialFingerprints: A dictionary where keys are versioned algorithm identifiers (e.g., "primaryLocationLineHash/v1") and values are components contributing to the fingerprint. Tools producing SARIF should populate this to aid consumers. GitHub uses primaryLocationLineHash.199
     + *Generation Strategy:* Combine rule ID, normalized location, and potentially code hashes to generate robust partial fingerprints.22 Avoid volatile elements like absolute paths or line numbers directly if possible.58
3. **Advanced Rule-Based Deduplication:** Go beyond simple fingerprint matching.27
   * *Superset/Subset Logic:* For findings related to dependency vulnerabilities, if Tool A reports CVE-1 in Lib v1.0 and Tool B reports CVE-1 in Lib v1.0.1 (which inherits the vulnerability), recognize these as related to the same underlying issue in the source library.
   * *Context-Aware Rules:* Define rules within the vulnerability management platform. Example: "If CWE-89 (SQLi) is found by Tool A in functionA() and Tool B also finds CWE-89 in functionA() (even on slightly different lines due to formatting), consider them duplicates." Another example: "Group all findings related to CVE-X across different microservices if they share the same vulnerable library version."
   * *Asset Context:* Link findings to specific assets (servers, containers, applications) using CMDB data. Deduplicate based on "Vulnerability Type + Asset" rather than just code location, especially for infrastructure or runtime findings.
4. **NLP/ML Approaches:** Utilize Natural Language Processing (NLP) and Machine Learning (ML) for semantic deduplication, especially useful when findings lack strong code-level identifiers or rely heavily on textual descriptions.259
   * *Techniques:*
     + *Text Preprocessing:* Clean and normalize finding descriptions (remove boilerplate, timestamps, etc.).
     + *Feature Extraction:* Convert text descriptions and metadata into numerical vectors using methods like TF-IDF or word/sentence embeddings (Word2Vec, GloVe, BERT, sentence-transformers 18).
     + *Similarity Calculation:* Compute similarity scores between finding vectors using metrics like cosine similarity.18
     + *Clustering:* Apply clustering algorithms (K-Means, DBSCAN 288) to group semantically similar findings. Findings within the same cluster are likely duplicates or closely related.
   * *Application:* Particularly useful for deduplicating findings from DAST tools, vulnerability reports, or threat intelligence feeds where code location is less precise or descriptions are the primary identifier. Can also augment fingerprinting by clustering findings with similar descriptions even if fingerprints differ slightly. Platforms like Faraday leverage deduplication 68, and DefectDojo has specific deduplication logic.27

Implementing a robust deduplication strategy requires combining stable fingerprinting based on vulnerability type and normalized location/code, potentially augmented by advanced rule-based logic and NLP/ML techniques for semantic similarity.

### **E. Vulnerability Location Tracking**

Maintaining accurate vulnerability location information despite code changes and refactoring is crucial for effective remediation and tracking.

1. **Challenges with Code Refactoring:** Simple line number-based location tracking is brittle. Renaming files, moving functions, adding/deleting lines, or reformatting code can cause SAST tools to lose track of previously identified vulnerabilities, leading to findings being incorrectly marked as "fixed" and then "new" again, or appearing at incorrect locations.297
2. **AST-Based Mapping:** Abstract Syntax Trees (ASTs) provide a more resilient way to map vulnerabilities.297
   * **How it Works:** Instead of mapping to a line number, the finding is mapped to a specific node or subtree within the AST representing the vulnerable code construct (e.g., a specific function call, variable assignment, class definition). Tools can parse the code, generate the AST, and locate the corresponding node.301
   * **Resilience:** AST-based mapping is more resistant to changes like adding/removing lines, comments, or whitespace, as long as the core structure of the vulnerable code segment remains intact within the AST. Refactoring that preserves the semantic structure (e.g., renaming a local variable within the function) is often handled well.
   * **Limitations:** Major refactoring that significantly alters the AST structure (e.g., extracting code into a new function, changing the type of node) can still break the mapping. AST generation itself can be complex and parser-dependent.301
3. **Other Techniques:**
   * **Logical Signatures/Fingerprinting:** As discussed in deduplication, creating fingerprints based on code structure, context, and vulnerability type (not just line number) inherently provides more resilient tracking.299 If the fingerprint remains the same across scans despite code changes, the vulnerability is considered the same instance.
   * **Semantic Diffing:** Tools can compare code versions based on semantic changes (AST diffing) rather than just textual diffs. This helps understand if a change actually fixed a vulnerability or just moved it.299
   * **Data Flow Analysis:** For taint-style vulnerabilities, tracking the data flow path from source to sink provides a more robust identifier than just the sink location.39 Even if the sink code moves, if the source-to-sink flow persists, the vulnerability likely still exists.

A combination of AST-based location mapping (where supported by the tool and SARIF representation) and robust fingerprinting offers the best approach to resilient vulnerability tracking through code evolution.

### **F. Data Storage Strategy**

Storing, querying, and correlating potentially large volumes of security findings from multiple tools and languages requires a well-designed data storage solution.

1. **Requirements:**
   * **Scalability:** Handle large volumes of findings generated over time from frequent CI/CD scans across many repositories.
   * **Query Performance:** Support fast retrieval for dashboards (e.g., findings per severity over time, findings per tool/language), trend analysis, and ad-hoc investigation queries.
   * **Correlation:** Efficiently query relationships between findings, vulnerabilities (CVEs), code components (files, functions), assets, and dependencies.
   * **Schema Flexibility:** Accommodate potentially diverse data structures from different tools, even after normalization to SARIF (e.g., custom properties fields).
   * **Data Security:** Ensure findings data is stored securely, with appropriate access controls and potentially encryption.309 Implement data retention policies.309
2. **Database Options Comparison:**
   * **PostgreSQL (Relational):**
     + *Pros:* Mature, ACID compliant, strong support for structured data and complex SQL queries (joins, aggregations). Good for enforcing data consistency. Can handle JSONB for semi-structured data. 311
     + *Cons:* Schema rigidity can be a challenge if finding formats vary greatly (though JSONB helps). Complex relationship queries (e.g., multi-hop dependency graphs) can require complex joins and may not perform as well as graph databases at scale.311 Handling very high ingest rates might require careful tuning.
     + *Use Case Fit:* Suitable if findings can be well-normalized into a relational schema or stored in JSONB. Good for structured querying, reporting, and trend analysis based on specific fields.
   * **Elasticsearch (Search Engine/Document Store):**
     + *Pros:* Excellent for full-text search on finding descriptions/messages. Highly scalable horizontally. Flexible schema (schemaless or explicit mapping) handles diverse data well.312 Strong aggregation capabilities for dashboards.312 Optimized for fast search and analysis of log-like data.312
     + *Cons:* Not ideal for complex relational queries involving joins.314 Eventual consistency model. Can require significant operational overhead for cluster management (though managed services exist).312 Less suited for enforcing strict transactional integrity compared to SQL databases.
     + *Use Case Fit:* Excellent for dashboards, searching finding details (text search), and aggregating metrics. Good fit if findings are treated as semi-structured documents. Performance depends heavily on mapping and query design.314
   * **Graph Databases (e.g., Neo4j):**
     + *Pros:* Natively designed for storing and querying relationships. Excels at traversing complex connections (e.g., finding all assets affected by vulnerabilities in a specific library, tracing attack paths).311 Flexible schema.315 Cypher query language is often intuitive for relationship queries.311
     + *Cons:* Can be more complex to implement and manage initially compared to relational databases.311 May not be as optimized for full-text search or simple aggregations across all nodes compared to Elasticsearch. Performance depends on the graph model design.
     + *Use Case Fit:* Ideal for advanced correlation analysis, understanding attack paths, visualizing dependencies between vulnerabilities, code components, and assets.317 Can complement other databases.
3. **Indexing Strategies:** Proper indexing is crucial for query performance, regardless of the chosen database.
   * **PostgreSQL:**
     + *B-tree (Default):* Best for equality and range queries on fields like finding\_id, tool\_name, severity (if ordered), cve, file\_path, line\_number.322
     + *Multicolumn Indexes:* Use for queries frequently filtering on multiple columns (e.g., (tool\_name, severity), (file\_path, line\_number)).322 Order matters.
     + *GIN:* Suitable for indexing array types or JSONB fields (e.g., fingerprint if it's complex, or SARIF properties).322 Can support full-text search.
     + *Hash:* Optimized for exact equality matches only.323
     + *BRIN:* Consider for very large tables with physically correlated data (e.g., timestamp).322
     + *Best Practices:* Index selective columns (high cardinality), use composite indexes for multi-column filters, consider expression indexes (e.g., LOWER(tool\_name)), avoid over-indexing (hurts writes), use CONCURRENTLY for index creation.322 Analyze query plans (EXPLAIN).322
   * **Elasticsearch:**
     + *Mapping:* Define explicit mappings for fields rather than relying solely on dynamic mapping.325 Use keyword type for exact matching/aggregation fields (finding\_id, tool\_name, severity, cve, fingerprint). Use appropriate numeric types (integer, float) for line\_number, scores. Use text (with appropriate analyzers) for full-text search fields (e.g., message, description).325
     + *Denormalization:* Avoid nested objects or parent-child relationships if possible, as they significantly slow down queries. Denormalize data by embedding relevant information within the finding document.314
     + *Index Design:* Optimize shard count and size based on data volume and query load. Too many small shards or too few large shards can hurt performance.314 Consider time-based indices (e.g., daily/weekly) for large volumes of findings to manage retention and optimize queries over specific time ranges.326
     + *Query Optimization:* Search as few fields as possible (use copy\_to for a general search field).314 Use filter context instead of query context for non-scoring queries. Use term queries on keyword fields. Optimize date histograms and aggregations.314 Use the Profile API to identify bottlenecks.314 Consider custom routing if queries often target findings related to a specific entity (e.g., asset ID).314
4. **Data Security and Retention:**
   * Implement access controls (role-based) to restrict access to findings data.309
   * Encrypt sensitive data at rest and in transit.309
   * Establish automated backup procedures (e.g., 3-2-1 method: 3 copies, 2 media types, 1 offsite).309
   * Define and enforce data retention policies based on compliance and operational needs.309
   * Utilize version control (like Git) for managing configurations and potentially scripts related to data handling.309

The choice of database depends on the primary access patterns. Elasticsearch excels at search and dashboarding, PostgreSQL offers relational integrity, and Graph Databases are superior for relationship analysis. A hybrid approach (e.g., Elasticsearch for dashboards, GraphDB for correlation) might be optimal for complex needs. Regardless of the choice, careful schema design and indexing are paramount for performance.

## **VI. Validation Strategy**

Validating the effectiveness of the multi-language security analysis strategy is crucial to ensure it meets its objectives for accuracy, coverage, and efficiency.

### **A. Selecting Test Repositories**

A diverse set of test repositories is needed to evaluate performance across different languages, frameworks, and vulnerability types.

1. **Intentionally Vulnerable Applications (IVAs):** Utilize well-maintained, deliberately insecure applications created for security training and testing. These provide known vulnerabilities to measure tool recall (True Positives).
   * *OWASP Juice Shop:* Node.js and Angular (Note: Not React as queried, but a valuable JS/Node.js target). Covers OWASP Top 10 extensively.328
   * *OWASP NodeGoat:* Node.js application (Inaccessible source 329, assumed suitable based on OWASP project goals).
   * *OWASP PyGoat:* Python/Django application covering OWASP Top Ten, MITRE CVEs, SANS Top 25.330
   * *OWASP WebGoat:* Java application covering OWASP Top 10.331
   * *(Consider) OWASP Security Knowledge Framework (SKF):* May contain labs for various languages/vulnerabilities, potentially including Scala or React.332
2. **Internal Applications:** Select representative internal applications or microservices that use the target languages (Scala, Node.js, Python, Java, React) and frameworks (Akka, Play, Express, NestJS, Django, Flask, Spring, Jakarta EE). These help validate the strategy's effectiveness on the organization's actual codebase and common patterns. Include projects with known historical vulnerabilities if possible.
3. **Framework-Specific Examples:** Include smaller code examples specifically demonstrating the frameworks targeted by custom rules or tool awareness checks (e.g., Akka cluster examples 334, Play starter templates 335, basic Express/Django apps).
4. **Mixed-Language Repositories:** Ensure test cases include repositories with a mix of the target languages to validate the language detection and orchestration logic.

### **B. Defining Validation Metrics**

Quantifiable metrics are needed to objectively assess the strategy's performance:

* **Detection Accuracy:**
  + *True Positive Rate (Recall):* Percentage of known vulnerabilities in the test repositories correctly identified by the toolchain. (TP / (TP + FN)).
  + *False Positive Rate:* Percentage of reported findings that are not actual vulnerabilities upon manual verification. (FP / (FP + TN)). 47
  + *False Negative Rate:* Percentage of known vulnerabilities missed by the toolchain. (FN / (TP + FN)).
  + *Overall Accuracy (e.g., Youden's Index):* A balanced measure combining sensitivity (Recall) and specificity (1 - FPR). Formula: Sensitivity + Specificity - 1.47 Useful for comparing tools with different TP/FP trade-offs.
* **Location Accuracy:** Measure how often the reported file and line number match the actual location of the vulnerability. Crucial for developer remediation.
* **Scan Performance:**
  + *Scan Duration:* Time taken to complete scans for each language/tool and the overall pipeline execution time. Measure against different repository sizes and types.
  + *Resource Consumption:* Monitor CPU and memory usage of scanner containers/processes during scans.
* **Normalization & Deduplication Effectiveness:**
  + *SARIF Compliance:* Validate generated SARIF reports against the schema.20
  + *Severity Mapping Consistency:* Verify that tool severities are consistently mapped to the internal scale.
  + *Deduplication Accuracy:* Measure how effectively the fingerprinting and deduplication logic identifies unique vulnerabilities across different tools and scans of the same code (e.g., before and after minor refactoring).

### **C. Validation Process**

Validation should be an iterative process:

1. **Establish Baseline:** Manually review and document the known vulnerabilities (True Positives) and expected non-vulnerabilities (True Negatives) within the selected test repositories. This forms the ground truth.
2. **Execute Strategy:** Run the complete analysis pipeline (language detection, tool orchestration, scanning, normalization, deduplication) against the test repositories.
3. **Collect Results:** Gather the normalized and deduplicated findings produced by the strategy.
4. **Compare and Calculate Metrics:** Compare the strategy's output against the established baseline to calculate the defined metrics (TP, FP, FN rates, accuracy, speed, etc.).
5. **Analyze Discrepancies:** Investigate false positives, false negatives, and location inaccuracies to identify weaknesses in tool configuration, custom rules, language detection, normalization, or deduplication logic.
6. **Iterate and Tune:** Adjust tool configurations, refine custom rules, improve fingerprinting logic, or update normalization mappings based on the analysis. Re-run the validation process to measure improvement.
7. **Regular Re-validation:** Periodically repeat the validation process (e.g., quarterly, annually, or after significant tool/rule updates) to ensure the strategy remains effective as tools evolve and new vulnerability patterns emerge.

This rigorous validation ensures the multi-language security analysis strategy is not just implemented but is demonstrably effective and efficient in identifying and managing vulnerabilities within the organization's specific context.

## **VII. Enhancing Findings and Prioritization**

Raw vulnerability findings, even when normalized and deduplicated, often lack the necessary context for effective prioritization. Enriching findings with data related to exploitability, code ownership, and business impact transforms a simple list of vulnerabilities into a risk-based action plan.

### **A. Contextual Enrichment**

Integrate data from various sources to add layers of context to each finding:

1. **Exploitability Data:** Determine the likelihood of a vulnerability being exploited in the wild.
   * **EPSS (Exploit Prediction Scoring System):** Programmatically query the EPSS API (336) using the CVE associated with a finding. Retrieve the EPSS score (probability of exploitation within 30 days) and percentile. Higher scores indicate greater urgency.336
   * **CISA KEV (Known Exploited Vulnerabilities):** Check if the finding's CVE exists in the CISA KEV catalog (339). A match signifies that the vulnerability is actively being exploited by threat actors and requires immediate attention.339 The KEV catalog can be downloaded or queried.
2. **Code Ownership and Churn:** Identify who owns the affected code and how recently it has changed, aiding remediation assignment and risk assessment.
   * **CODEOWNERS Files:** Parse CODEOWNERS files within the repository to map file paths or patterns associated with a finding to the responsible teams or individuals.337 This requires access to the repository structure.
   * **git blame:** For findings with precise code locations, execute git blame --porcelain <file> or git blame --line-porcelain <file> programmatically.343 Parse the output to extract the commit hash, author name, author email, and commit timestamp for the specific line(s) of the finding.343 This identifies the last modifier and the age of the vulnerable code, which can indicate code stability or recent churn.337
3. **Asset and Business Context:** Understand the importance and exposure of the asset affected by the finding.
   * **CMDB/Asset Inventory:** Integrate with the organization's CMDB (e.g., ServiceNow 344) or asset management system. Query the system using asset identifiers (e.g., hostname, IP address, application name, repository name) linked to the finding's location or deployment target. Retrieve attributes like:
     + *Business Criticality:* (e.g., High, Medium, Low).344
     + *Environment:* (e.g., Production, Staging, Development, Test).344
     + *Data Sensitivity:* Does the asset handle PII, financial data, credentials, etc.?.239
     + *Asset Owner:* Responsible team or individual.344
   * **Network Context:** Determine the asset's exposure.
     + *Internet Exposure:* Is the affected asset directly or indirectly exposed to the internet?.239
     + *Access to Critical Resources:* Does the potentially compromised asset have network access or credentials to other high-value systems (e.g., databases)?.239

### **B. Integrating Enrichment Data**

The process of adding context should be automated:

1. **Data Aggregation Point:** Enrichment should occur after findings are normalized (e.g., into SARIF) and ideally stored in a central vulnerability management database or platform.
2. **API Queries and Scripting:** Develop scripts or leverage platform features to:
   * Extract identifiers (CVE, file path, asset name) from normalized findings.
   * Query external APIs (EPSS 336, CISA KEV 339, CMDB/Asset Inventory 344).
   * Execute local commands (git blame 343).
   * Parse CODEOWNERS files.
   * Update the finding records in the central store with the retrieved contextual data (e.g., adding custom fields or populating dedicated attributes).

### **C. Advanced Prioritization**

Move beyond simple severity-based prioritization by implementing a risk-based scoring model that incorporates the enriched context.

1. **Risk Score Calculation:** Define a formula or matrix that combines multiple factors:
   * **Normalized Severity:** The finding's severity mapped to the internal scale (Section V.B).
   * **Exploitability:** High weight for KEV status.339 Incorporate EPSS score (e.g., higher score = higher risk factor).336
   * **Business Impact:** Assign weights based on asset criticality, data sensitivity, and environment (Production > Staging > Dev).239
   * **Contextual Modifiers:** Increase risk score based on factors like internet exposure 239, access to other critical resources 239, or recent code churn.343 Decrease score for mitigating factors identified in VEX (not\_affected state) 60 or for findings on non-critical dev assets.
2. **Prioritization Logic:** Rank findings based on the calculated risk score. Address findings with the highest risk scores first.
3. **Remediation Routing:** Use the identified Code Owner information to automatically assign or suggest assignees for remediation tasks within the vulnerability management platform or ticketing system.337

This contextual, risk-based approach ensures that remediation efforts are focused on the vulnerabilities posing the greatest actual threat to the organization.239

### **D. Integration with Vulnerability Management Platforms**

A central platform is essential for managing the entire lifecycle of enriched findings.

1. **Platform Choice:** Select a platform capable of ingesting normalized findings (ideally SARIF), storing custom enrichment data, supporting custom risk scoring, and facilitating remediation workflows.
   * *OWASP DefectDojo:* An open-source option supporting numerous scanner imports, deduplication, reporting, and JIRA integration.27 Its data model is flexible.67 Deduplication logic helps manage duplicates.27 Offers API for integration.347
   * *Faraday:* A commercial platform emphasizing integration (150+ tools), normalization, deduplication, automation (Agents, Workflows), and collaboration.68
   * *Other Platforms:* Consider other commercial or open-source ASPM (Application Security Posture Management) tools.
2. **Workflow Integration:**
   * Use the platform's API 347 or built-in features to automatically import enriched and prioritized findings.
   * Configure workflows to create remediation tickets (e.g., in Jira 66) automatically based on risk score and code ownership.
   * Track remediation progress and finding status within the platform.
   * Generate dashboards and reports based on the enriched data for stakeholder communication.

## **VIII. Conclusion and Recommendations**

### **A. Summary of the Multi-Language Strategy**

This report has detailed a comprehensive strategy for implementing security analysis across diverse programming languages, specifically Scala, Node.js, Python, Java, and React. The core components of this strategy involve:

1. **Accurate Language Detection:** Employing a multi-faceted approach using file extensions, content analysis, and specialized tools like Linguist or enry to correctly identify languages at the file level.
2. **Layered Tool Selection:** Choosing a suite of SAST, SCA, and secret scanning tools based on rigorous criteria including language/framework support, rule quality, integration capabilities, performance, and output format standardization (SARIF).
3. **Targeted Configuration:** Utilizing custom rules and tool configurations to enhance framework awareness, address organization-specific concerns, and manage false positives effectively.
4. **Automated Orchestration:** Integrating the toolchain into CI/CD pipelines, leveraging containerization for consistency, and optimizing execution speed through parallelization and caching.
5. **Robust Output Processing:** Normalizing diverse tool outputs to the SARIF standard, leveraging VEX for exploitability context, implementing a consistent internal severity scale (using CVSS as a baseline), managing false positives through suppression and feedback loops (including confidence scoring), and employing robust fingerprinting techniques (code attributes, AST, NLP) for effective deduplication.
6. **Contextual Prioritization:** Enriching normalized findings with exploitability data (EPSS, KEV), code context (CODEOWNERS, git blame), and asset/business information (CMDB) to enable risk-based prioritization.
7. **Centralized Management:** Utilizing a vulnerability management platform (like DefectDojo or Faraday) to aggregate findings, track remediation, and generate reports.
8. **Continuous Validation:** Regularly validating the entire strategy using representative test repositories and defined metrics to ensure ongoing effectiveness.

This holistic approach moves beyond simple vulnerability scanning towards a mature, risk-informed security analysis program capable of handling the complexities of modern, polyglot software development.

### **B. Key Implementation Recommendations**

Successfully implementing this strategy requires a phased and systematic approach:

1. **Pilot Project:** Begin by implementing the strategy for a single application or a small set of applications representing the target language mix. This allows for refining processes and tool configurations in a controlled environment.
2. **Invest in Central Platform:** Select and implement a central vulnerability management platform early. Its capabilities for ingestion, normalization, deduplication, enrichment, and workflow automation are critical for scaling the strategy.66 Ensure the platform supports SARIF import and allows for custom data fields and risk scoring.
3. **Standardize Outputs:** Mandate SARIF 2.1.0 as the standard output format for all integrated security tools. Develop transformers for tools lacking native support. Utilize CycloneDX VEX for communicating exploitability for SCA findings.
4. **Develop Internal Standards:** Document the internal severity normalization methodology, mapping tool outputs and CVSS scores to the defined internal scale. Define the risk prioritization model incorporating contextual factors.
5. **Establish Clear Processes:** Define workflows for:
   * Tool onboarding and configuration.
   * Custom rule development, testing, and management.
   * False positive triage, suppression, and feedback loop implementation.
   * Remediation assignment and tracking.
6. **Prioritize Automation:** Automate language detection, tool execution within CI/CD, output normalization, data enrichment queries (APIs, git commands), and basic reporting as much as possible.
7. **Tool Integration:** Focus on integrating tools seamlessly into the developer workflow (IDE plugins, CI/CD checks) and the central vulnerability management platform.
8. **Training:** Provide training to development and security teams on the selected tools, the defined processes (triage, remediation), secure coding practices relevant to findings, and the importance of the feedback loop.
9. **Iterative Rollout:** Gradually expand the strategy to cover more applications and teams, incorporating lessons learned from the pilot phase.

### **C. Future Considerations**

The landscape of application security is constantly evolving. Future enhancements to this strategy could include:

1. **AI-Assisted Triage and Remediation:** Further leverage AI/ML not only for confidence scoring and deduplication 293 but also for automated triage suggestions, root cause analysis, and potentially generating code fixes.15
2. **Deeper Supply Chain Security:** Expand beyond basic SCA to incorporate more comprehensive supply chain security measures, such as verifying artifact provenance using tools like Sigstore/Cosign 357 and analyzing build process integrity using frameworks like in-toto.359 Integrate SBOM analysis more deeply into risk assessment.
3. **Enhanced Contextualization:** Incorporate more runtime context (e.g., from DAST or IAST tools, runtime protection agents) to further refine exploitability analysis and prioritization.
4. **Adapting to New Technologies:** Continuously evaluate and adapt the toolchain and rulesets to support new programming languages, frameworks, and architectural patterns (e.g., WebAssembly, serverless) as they are adopted by the organization.
5. **Unified Security Data Lake:** Consider consolidating security findings and enrichment data into a centralized data lake alongside other security telemetry (logs, network data) to enable broader threat hunting and correlation capabilities, potentially using technologies like Elasticsearch or specialized security data platforms.

By adopting the outlined strategy and remaining adaptable to future developments, organizations can significantly improve their ability to manage security risks effectively across complex, multi-language software portfolios.

#### Works cited

1. Managing State – React, accessed May 8, 2025, <https://react.dev/learn/managing-state#choosing-the-state-structure>
2. Rules Sonarsource, accessed May 8, 2025, <https://rules.sonarsource.com/>
3. Code Language Detector · Actions · GitHub Marketplace, accessed May 8, 2025, <https://github.com/marketplace/actions/code-language-detector>
4. pemistahl/lingua-py: The most accurate natural language detection library for Python, suitable for short text and mixed-language text - GitHub, accessed May 8, 2025, <https://github.com/pemistahl/lingua-py>
5. Top Static Application Security Testing (SAST) Software for Scala in 2025 - Slashdot, accessed May 8, 2025, <https://slashdot.org/software/static-application-security-testing-sast/for-scala/>
6. Best Static Application Security Testing (SAST) Software for Node.js - Slashdot, accessed May 8, 2025, <https://slashdot.org/software/static-application-security-testing-sast/for-node.js/>
7. Python SAST Tools: Free & Paid Solutions for Secure Code Analysis - Bito AI, accessed May 8, 2025, <https://bito.ai/blog/python-sast-tools/>
8. The 23 Best Static Code Analysis Tools for Java of 2025 - The CTO Club, accessed May 8, 2025, <https://thectoclub.com/tools/best-static-code-analysis-tools-java/>
9. Any good SAST tools for React specifically? - Reddit, accessed May 8, 2025, <https://www.reddit.com/r/react/comments/1izxzc4/any_good_sast_tools_for_react_specifically/>
10. Source Code Analysis Tools | OWASP Foundation, accessed May 8, 2025, <https://owasp.org/www-community/Source_Code_Analysis_Tools>
11. Engine Pack Supported Code Languages and Frameworks (9.7.0), accessed May 8, 2025, <https://docs.checkmarx.com/en/34965-325334-engine-pack-supported-code-languages-and-frameworks--9-7-0-.html>
12. Documentation | NestJS - A progressive Node.js framework, accessed May 8, 2025, <https://docs.nestjs.com/>
13. SAST Scanner - Supported Languages and Frameworks - Checkmarx Documentation, accessed May 8, 2025, <https://docs.checkmarx.com/en/34965-324495-sast-scanner---supported-languages-and-frameworks.html>
14. Jakarta EE vs. Spring Boot: Choosing the Right Framework for Your Project - Payara Blog, accessed May 8, 2025, <https://blog.payara.fish/jakarta-ee-vs.-spring-boot-choosing-the-right-framework-for-your-project>
15. Semgrep Code, accessed May 8, 2025, <https://semgrep.dev/products/semgrep-code/>
16. Advanced security with SonarQube | Sonar, accessed May 8, 2025, <https://www.sonarsource.com/solutions/security/>
17. 13 Best Static Code Analysis Tools For 2025 - Qodo, accessed May 8, 2025, <https://www.qodo.ai/blog/best-static-code-analysis-tools/>
18. Autogrep: Automated Generation and Filtering of Semgrep Rules from Vulnerability Patches, accessed May 8, 2025, <https://lambdasec.github.io/AutoGrep-Automated-Generation-and-Filtering-of-Semgrep-Rules-from-Vulnerability-Patches/>
19. Static Analysis Results Interchange Format (SARIF) Version 2.1.0, accessed May 8, 2025, <https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html>
20. sarif-schema-2.1.0.json - GitHub, accessed May 8, 2025, <https://github.com/oasis-tcs/sarif-spec/blob/main/sarif-2.1/schema/sarif-schema-2.1.0.json>
21. What is Static Analysis Results Interchange Format (SARIF)? | Aptori ..., accessed May 8, 2025, <https://www.aptori.com/glossary/static-analysis-results-interchange-format-sarif>
22. SARIF Can't Save You Now - Boost Security, accessed May 8, 2025, <https://boostsecurity.io/blog/sarif-cant-save-you-now>
23. False Positive Elimination: Enhancing Cybersecurity Efficiency ..., accessed May 8, 2025, <https://brandefense.io/blog/drps/false-positive-elimination-enhancing-cybersecurity-efficiency/>
24. Bandit Documentation, accessed May 8, 2025, <https://bandit.readthedocs.io/_/downloads/en/1.7.5/pdf/>
25. Frequently Asked Questions | SonarQube Docs, accessed May 8, 2025, <https://pmpl.cs.ui.ac.id/sonarqube/documentation/faq/>
26. Triage and remediate findings - Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-code/triage-remediation>
27. About Deduplication | DefectDojo Documentation, accessed May 8, 2025, <https://docs.defectdojo.com/en/working_with_findings/finding_deduplication/about_deduplication/>
28. Semgrep extension for Visual Studio Code - GitHub, accessed May 8, 2025, <https://github.com/semgrep/semgrep-vscode>
29. Analysis parameters | SonarQube Server Documentation, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube-server/10.8/analyzing-source-code/analysis-parameters/>
30. linguist/lib/linguist/languages.yml at main · github-linguist/linguist ..., accessed May 8, 2025, <https://github.com/github-linguist/linguist/blob/master/lib/linguist/languages.yml>
31. github-linguist/linguist: Language Savant. If your ... - GitHub, accessed May 8, 2025, <https://github.com/github-linguist/linguist>
32. go-enry/go-enry: A faster file programming language detector - GitHub, accessed May 8, 2025, <https://github.com/go-enry/go-enry>
33. C# or Java? TypeScript or JavaScript? Machine learning based classification of programming languages - The GitHub Blog, accessed May 8, 2025, <https://github.blog/ai-and-ml/machine-learning/c-or-java-typescript-or-javascript-machine-learning-based-classification-of-programming-languages/>
34. Component Analysis | OWASP Foundation, accessed May 8, 2025, <https://owasp.org/www-community/Component_Analysis>
35. spotbugs Documentation, accessed May 8, 2025, <https://spotbugs.readthedocs.io/_/downloads/en/issue-1075/pdf/>
36. A03 Injection - OWASP Top 10:2021, accessed May 8, 2025, <https://owasp.org/Top10/A03_2021-Injection/>
37. Rules | SonarQube Server Documentation, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube/latest/user-guide/rules/>
38. Find Security Bugs, accessed May 8, 2025, <https://find-sec-bugs.github.io/>
39. Rule syntax | Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/writing-rules/rule-syntax/#taint-tracking>
40. accessed December 31, 1969, <https://docs.sonarsource.com/sonarqube/latest/analysis/security-rules/#taint-vulnerabilities>
41. Futures and Promises | Scala Documentation, accessed May 8, 2025, <https://docs.scala-lang.org/overviews/core/futures.html#common-pitfalls>
42. accessed December 31, 1969, <https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-secure-coding-practices-java/java-reflection>
43. Unveiling the Prototype Pollution Gadgets Finder - Doyensec's Blog, accessed May 8, 2025, <https://blog.doyensec.com/2024/02/17/server-side-prototype-pollution-Gadgets-scanner.html>
44. Object - JavaScript | MDN, accessed May 8, 2025, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype#security_risks_of_prototype_pollution>
45. How to Setup Semgrep Rules for Optimal SAST Scanning - Jit.io, accessed May 8, 2025, <https://www.jit.io/resources/appsec-tools/semgrep-rules-for-sast-scanning>
46. Cross Site Scripting Prevention - OWASP Cheat Sheet Series, accessed May 8, 2025, <https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#react-dom-rendering-options>
47. Evaluating SAST Tools | Praetorian, accessed May 8, 2025, <https://www.praetorian.com/blog/evaluating-sast-tools/>
48. Best Static Application Security Testing (SAST) Software: User ... - G2, accessed May 8, 2025, <https://www.g2.com/categories/static-application-security-testing-sast>
49. accessed December 31, 1969, <https://www.g2.com/categories/software-composition-analysis-sca>
50. accessed December 31, 1969, <https://www.capterra.com/application-security-software/>
51. Scala | SonarQube Server Documentation, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/languages/scala/>
52. accessed December 31, 1969, <https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/languages/javascript-typescript/>
53. Python | SonarQube Server Documentation, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/languages/python/>
54. Java | SonarQube Server Documentation, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/languages/java/>
55. TSX - ast-grep, accessed May 8, 2025, <https://ast-grep.github.io/catalog/tsx/>
56. Beyond Benchmarks: How Semgrep Redefines Javascript Security, accessed May 8, 2025, <https://semgrep.dev/blog/2025/beyond-benchmarks-how-semgrep-redefines-javascript-security>
57. Static Application Security Testing (SAST) - GitLab Docs, accessed May 8, 2025, <https://docs.gitlab.com/user/application_security/sast/>
58. Static Analysis Results Interchange Format (SARIF) Version 2.1.0, accessed May 8, 2025, <https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html#_Toc34317648>
59. SARIF support for code scanning - GitHub Docs, accessed May 8, 2025, <https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning>
60. How CycloneDX VEX Makes Your SBOM Useful | Blog - Endor Labs, accessed May 8, 2025, <https://www.endorlabs.com/learn/how-cyclonedx-vex-makes-your-sbom-useful>
61. Security Use Case: Vulnerability Exploitability - CycloneDX, accessed May 8, 2025, <https://cyclonedx.org/use-cases/vulnerability-exploitability/>
62. Vulnerability Exploitability eXchange (VEX) - CycloneDX, accessed May 8, 2025, <https://cyclonedx.org/capabilities/vex/>
63. Anchore Container Scan · Actions · GitHub Marketplace, accessed May 8, 2025, <https://github.com/marketplace/actions/anchore-container-scan>
64. CI/CD Security Scanning: Types & Best Practices - SentinelOne, accessed May 8, 2025, <https://www.sentinelone.com/cybersecurity-101/cloud-security/ci-cd-security-scanning/>
65. SonarQube for IDE | Eclipse Plugins, Bundles and Products, accessed May 8, 2025, <https://marketplace.eclipse.org/content/sonarqube-ide>
66. DefectDojo/django-DefectDojo: DevSecOps, ASPM, Vulnerability Management. All on one platform. - GitHub, accessed May 8, 2025, <https://github.com/DefectDojo/django-DefectDojo>
67. OWASP Defectdojo, accessed May 8, 2025, <https://owasp.org/www-project-defectdojo/>
68. Faraday Security - Protect your business, scale your security., accessed May 8, 2025, <https://faradaysec.com/>
69. Configure a ImageVulnerabilityScan for Grype - TechDocs, accessed May 8, 2025, <https://techdocs.broadcom.com/us/en/vmware-tanzu/standalone-components/tanzu-application-platform/1-12/tap/scst-scan-ivs-grype.html>
70. Bandwidth bandit: Understanding memory contention - IEEE Computer Society, accessed May 8, 2025, <https://www.computer.org/csdl/proceedings-article/ispass/2012/06189214/12OmNx5piTn>
71. How Syft Scans Software to Generate SBOMs - Anchore, accessed May 8, 2025, <https://anchore.com/blog/how-syft-scans-software-to-generate-sboms/>
72. Semgrep Network Broker, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-ci/network-broker>
73. CONTRIBUTING.md - trailofbits/semgrep-rules - GitHub, accessed May 8, 2025, <https://github.com/trailofbits/semgrep-rules/blob/main/CONTRIBUTING.md>
74. Pre-optimize all TM grammars for performance · Issue #7295 - GitHub, accessed May 8, 2025, <https://github.com/github-linguist/linguist/issues/7295>
75. How to Run a SAST Test with Bandit and Jit - Jit.io, accessed May 8, 2025, <https://www.jit.io/resources/appsec-tools/how-to-run-a-sast-test-with-bandit-and-jit>
76. The best static analysis tools and linters for Java - Symflower, accessed May 8, 2025, <https://symflower.com/en/company/blog/2024/static-analysis-java/>
77. SAST Scanner Parameters - Checkmarx Documentation, accessed May 8, 2025, <https://docs.checkmarx.com/en/34965-324305-sast-scanner-parameters.html>
78. Command Line Interface Reference - ESLint - Pluggable JavaScript Linter, accessed May 8, 2025, <https://eslint.org/docs/latest/use/command-line-interface>
79. Custom rules - Semgrep Secrets, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-secrets/rules>
80. Adding coding rules | SonarQube Server Documentation, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube-server/latest/extension-guide/adding-coding-rules/>
81. JavaScript | SonarQube Docs, accessed May 8, 2025, <https://pmpl.cs.ui.ac.id/sonarqube/documentation/analysis/languages/javascript/>
82. Adding Coding Rules | SonarQube Docs, accessed May 8, 2025, <https://scm.thm.de/sonar/documentation/extend/adding-coding-rules/>
83. sonar-java/docs/CUSTOM\_RULES\_101.md at master - GitHub, accessed May 8, 2025, <https://github.com/SonarSource/sonar-java/blob/master/docs/CUSTOM_RULES_101.md>
84. Custom Rule Tutorial - ESLint - Pluggable JavaScript Linter, accessed May 8, 2025, <https://eslint.org/docs/latest/extend/custom-rule-tutorial>
85. gitleaks/gitleaks: Find secrets with Gitleaks - GitHub, accessed May 8, 2025, <https://github.com/gitleaks/gitleaks>
86. trufflesecurity/trufflehog: Find, verify, and analyze leaked ... - GitHub, accessed May 8, 2025, <https://github.com/trufflesecurity/trufflehog>
87. Scala | SonarQube Server Documentation, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube-server/latest/analysis/languages/scala/>
88. Supported languages - Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/supported-languages>
89. OWASP Dependency-Check, accessed May 8, 2025, <https://owasp.org/www-project-dependency-check/>
90. Checkmarx SCA Resolver Changelog, accessed May 8, 2025, <https://docs.checkmarx.com/en/34965-336728-checkmarx-sca-resolver-changelog.html>
91. accessed December 31, 1969, <https://docs.snyk.io/scan-with-snyk/snyk-open-source/snyk-open-source-language-and-package-manager-support>
92. accessed December 31, 1969, <https://docs.veracode.com/r/Supported_Languages_and_Platforms>
93. accessed December 31, 1969, <https://www.mend.io/docs/sca-languages-and-package-managers/>
94. JavaScript | Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/languages/javascript>
95. Run rules - Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/running-rules>
96. Getting Started with ESLint - ESLint - Pluggable JavaScript Linter, accessed May 8, 2025, <https://eslint.org/docs/latest/use/getting-started>
97. Core Concepts - ESLint - Pluggable JavaScript Linter, accessed May 8, 2025, <https://eslint.org/docs/latest/use/core-concepts/>
98. JavaScript static analysis comparison: ESLint vs Semgrep, accessed May 8, 2025, <https://semgrep.dev/blog/2021/javascript-static-analysis-comparison-eslint-semgrep/>
99. Configure Language Options - ESLint - Pluggable JavaScript Linter, accessed May 8, 2025, <https://eslint.org/docs/latest/user-guide/configuring/language-options>
100. Custom Rules - typescript-eslint, accessed May 8, 2025, <https://typescript-eslint.io/developers/custom-rules/>
101. Updates | Veracode Docs, accessed May 8, 2025, <https://docs.veracode.com/updates>
102. accessed December 31, 1969, <https://docs.snyk.io/scan-with-snyk/snyk-code/snyk-code-language-and-framework-support>
103. accessed December 31, 1969, <https://www.mend.io/docs/sast-supported-languages/>
104. accessed December 31, 1969, <https://owasp.org/www-project-dependency-check/latest/dependency-check-npm/index.html>
105. OWASP Dependency Check in Node js 🛡️ - DEV Community, accessed May 8, 2025, <https://dev.to/silentwatcher_95/owasp-dependency-check-in-node-js-1oo6>
106. Best Software Composition Analysis (SCA) Tools for Node.js - SourceForge, accessed May 8, 2025, <https://sourceforge.net/software/software-composition-analysis-sca/integrates-with-node.js/>
107. SCA Scanner - Supported Languages and Package Managers - Checkmarx Documentation, accessed May 8, 2025, <https://docs.checkmarx.com/en/34965-322679-sca-scanner---supported-languages-and-package-managers.html>
108. Python | Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/languages/python>
109. Manage rules and policies - Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-code/policies>
110. Bandit | Precaution, accessed May 8, 2025, <https://docs.securesauce.dev/comparisons/bandit>
111. Test Plugins — Bandit documentation, accessed May 8, 2025, <https://bandit.readthedocs.io/en/latest/plugins/>
112. bandit.plugins package - OpenStack Docs, accessed May 8, 2025, <https://docs.openstack.org/developer/bandit/api/bandit.plugins.html>
113. OWASP Dependency-Check | Software Testing Glossary - Parasoft, accessed May 8, 2025, <https://www.parasoft.com/glossary/owasp-dependency-check/>
114. accessed December 31, 1969, <https://owasp.org/www-project-dependency-check/latest/dependency-check-python/index.html>
115. Python | Snyk User Docs, accessed May 8, 2025, <https://docs.snyk.io/supported-languages-package-managers-and-frameworks/python>
116. Pro rules - Semgrep, accessed May 8, 2025, <https://semgrep.dev/products/semgrep-code/pro-rules/>
117. 30 new Semgrep rules: Ansible, Java, Kotlin, shell scripts, and more - The Trail of Bits Blog, accessed May 8, 2025, <https://blog.trailofbits.com/2024/01/17/30-new-semgrep-rules-ansible-java-kotlin-shell-scripts-and-more/>
118. Using the SpotBugs Maven Plugin — spotbugs 4.9.3 documentation, accessed May 8, 2025, <https://spotbugs.readthedocs.io/en/latest/maven.html>
119. Custom FindBugs detectors in Android - rhye.org, accessed May 8, 2025, <https://rhye.org/post/custom-android-findbugs/>
120. accessed December 31, 1969, <https://spotbugs.readthedocs.io/en/latest/writing_detectors.html>
121. Overview :: Jakarta EE Tutorial, accessed May 8, 2025, <https://jakarta.ee/learn/docs/jakartaee-tutorial/current/intro/overview/overview.html>
122. OWASP Developer Guide | Dependency-Check, accessed May 8, 2025, <https://owasp.org/www-project-developer-guide/draft/implementation/dependencies/dependency_check/>
123. accessed December 31, 1969, <https://owasp.org/www-project-dependency-check/latest/dependency-check-maven/index.html>
124. SCA Configuration Options - Checkmarx Documentation, accessed May 8, 2025, <https://docs.checkmarx.com/en/34965-349223-sca-configuration-options.html>
125. Write cleaner React code with SonarQube 10.4 | Sonar, accessed May 8, 2025, <https://www.sonarsource.com/blog/clean-react-code-sonarqube/>
126. Semgrep Pro rules, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-code/pro-rules>
127. eslint-plugin-react-compiler - Yarn 1, accessed May 8, 2025, <https://classic.yarnpkg.com/en/package/eslint-plugin-react-compiler>
128. reactjs - Safe alternative to dangerouslySetInnerHTML - Stack Overflow, accessed May 8, 2025, <https://stackoverflow.com/questions/29044518/safe-alternative-to-dangerouslysetinnerhtml>
129. React Security: Vulnerabilities & Best Practices [2025 Edit] - Glorywebs, accessed May 8, 2025, <https://www.glorywebs.com/blog/react-security-practices>
130. OWASP Dependency Check found 83 Critical Security Vulnerabilities in react-scripts:4.0.1, 3.4.4 package #10323 - GitHub, accessed May 8, 2025, <https://github.com/facebook/create-react-app/issues/10323>
131. Pro engine - Semgrep, accessed May 8, 2025, <https://semgrep.dev/products/pro-engine/>
132. Security related rules | SonarQube Cloud Documentation, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube-cloud/digging-deeper/security-related-rules/>
133. FloQast addresses security issues in minutes using Semgrep, accessed May 8, 2025, <https://www.semgrep.dev/case-studies/floqast-semgrep-code/>
134. Security in Django | Django documentation | Django, accessed May 8, 2025, <https://docs.djangoproject.com/en/5.0/topics/security/#template-system-security>
135. Custom Rules - ESLint - Pluggable JavaScript Linter, accessed May 8, 2025, <https://eslint.org/docs/latest/developer-guide/working-with-rules>
136. How to introduce Semgrep to your organization - The Trail of Bits Blog, accessed May 8, 2025, <https://blog.trailofbits.com/2024/01/12/how-to-introduce-semgrep-to-your-organization/>
137. Automating Security Workflows with the Semgrep Policy Management API, accessed May 8, 2025, <https://semgrep.dev/blog/2025/automating-security-workflows-with-the-semgrep-policy-management-api/>
138. OWASP DevSecOps Guideline | OWASP Foundation, accessed May 8, 2025, <https://owasp.org/www-project-devsecops-guideline/>
139. pivotal/LicenseFinder: Find licenses for your project's dependencies. - GitHub, accessed May 8, 2025, <https://github.com/pivotal/LicenseFinder>
140. license\_finder - Versions diffs - 7.0.1 → 7.1.0 - RubyGems - Mend, accessed May 8, 2025, <https://my.diffend.io/gems/license_finder/prev/7.1.0>
141. license\_finder - Versions diffs - 3.0.2 → 3.0.4 - RubyGems - Mend, accessed May 8, 2025, <https://my.diffend.io/gems/license_finder/prev/3.0.4>
142. Files · add-pnpm-support - license-finder Community Fork - GitLab, accessed May 8, 2025, <https://gitlab.com/gitlab-community/security-products/analyzers/license-finder/-/tree/add-pnpm-support>
143. Bandit - PyPI, accessed May 8, 2025, <https://pypi.org/project/bandit/>
144. semgrep-cli contributing, accessed May 8, 2025, <https://semgrep.dev/docs/contributing/semgrep-contributing>
145. openComplianceCode/scancode-toolkit - Gitee, accessed May 8, 2025, <https://gitee.com/openComplianceCode/scancode-toolkit>
146. scancode-toolkit - PyPI, accessed May 8, 2025, <https://pypi.org/project/scancode-toolkit/>
147. semgrep/semgrep - Docker Image | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/r/semgrep/semgrep>
148. Packages in the Semgrep Docker image, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-ci/packages-in-semgrep-docker>
149. Image Layer Details - semgrep/semgrep:1.79-nonroot | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/semgrep/semgrep/1.79-nonroot/images/sha256-4632b99be4815dcda9465aed37877fb11bd9c2dfe60e7c7431260dfe847ab75d>
150. Image Layer Details - semgrep/semgrep:latest-nonroot - Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/semgrep/semgrep/latest-nonroot/images/sha256-4ff9d4521e76021aa207c5f0b4963f5699bee5c780ac290e786ce2905cad09a0>
151. PyCQA/bandit: Bandit is a tool designed to find common security issues in Python code., accessed May 8, 2025, <https://github.com/PyCQA/bandit>
152. bandit/bandit versions · pycqa - GitHub, accessed May 8, 2025, <https://github.com/orgs/PyCQA/packages/container/bandit%2Fbandit/331203132?tag=latest>
153. secfigo/bandit - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/secfigo/bandit>
154. opensorcery/bandit - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/opensorcery/bandit>
155. pipelinecomponents/eslint - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/pipelinecomponents/eslint>
156. Resolving eslint Issues in Projects - Layer5 Community Forum, accessed May 8, 2025, <https://discuss.layer5.io/t/resolving-eslint-issues-in-projects/2365>
157. 10 best practices to containerize Node.js web applications with Docker | Snyk Blog, accessed May 8, 2025, <https://snyk.io/blog/10-best-practices-to-containerize-nodejs-web-applications-with-docker/>
158. Image Layer Details - warasw/tep:spotbugs | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/warasw/tep/spotbugs/images/sha256-b36bb69a593970778abc6aaab88a972822671b9fadac24546f6216d2787a4cd8>
159. Introduction – spotbugs-maven-plugin, accessed May 8, 2025, <https://spotbugs.github.io/spotbugs-maven-plugin/>
160. gradle - Official Image - Docker Hub, accessed May 8, 2025, <https://hub.docker.com/_/gradle>
161. SpotBugs, accessed May 8, 2025, <https://spotbugs.github.io/>
162. shibme/findsecbugs-docker - GitHub, accessed May 8, 2025, <https://github.com/shibme/findsecbugs-docker>
163. spotbugs:check – spotbugs-maven-plugin, accessed May 8, 2025, <https://spotbugs.github.io/spotbugs-maven-plugin/check-mojo.html>
164. Integrating findsecbugs plugin with Gradle's findbugs plugin - Stack Overflow, accessed May 8, 2025, <https://stackoverflow.com/questions/34085928/integrating-findsecbugs-plugin-with-gradles-findbugs-plugin>
165. sonar-scanner-cli-docker/Dockerfile at master - GitHub, accessed May 8, 2025, <https://github.com/SonarSource/sonar-scanner-cli-docker/blob/master/Dockerfile>
166. SonarScanner CLI | SonarQube Server Documentation, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube-server/10.8/analyzing-source-code/scanners/sonarscanner/>
167. Installing from Docker | SonarQube Docs, accessed May 8, 2025, <https://docs.sonarsource.com/sonarqube-server/10.7/setup-and-upgrade/install-the-server/installing-sonarqube-from-docker/>
168. sonarsource/sonar-scanner-cli - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/sonarsource/sonar-scanner-cli>
169. org.owasp.dependencycheck - Gradle Plugin Portal, accessed May 8, 2025, <https://plugins.gradle.org/plugin/org.owasp.dependencycheck>
170. accessed December 31, 1969, <https://owasp.org/www-project-dependency-check/latest/sbt/index.html>
171. anchore/syft - Docker Image | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/r/anchore/syft>
172. BenjiTrapp/syft-fork: CLI tool and library for generating a Software Bill of Materials from container images and filesystems - GitHub, accessed May 8, 2025, <https://github.com/BenjiTrapp/syft-fork>
173. Image Layer Details - anchore/syft:v0.91.0 | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/anchore/syft/v0.91.0/images/sha256-5bf41ee55321901e89dc2314328130046cb09ef937bcdb6fee810b0f5dc7cbe8>
174. Image Layer Details - anchore/syft:v1.23.1 | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/anchore/syft/v1.23.1/images/sha256-2e5ae1f1355c0c7181fdf4745b91d772001a3e7cb5b67717594b12bfde67db29>
175. syft - overview - Chainguard Containers, accessed May 8, 2025, <https://images.chainguard.dev/directory/image/syft/overview>
176. anchore/grype: A vulnerability scanner for container images ... - GitHub, accessed May 8, 2025, <https://github.com/anchore/grype>
177. Open Source Vulnerability Scanning: Tools & How-To Guide - Anchore, accessed May 8, 2025, <https://anchore.com/software-supply-chain-security/open-source-container-vulnerability-scanning-tools/>
178. anchore/grype - Docker Image | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/r/anchore/grype>
179. Dockerfile - anchore/grype - GitHub, accessed May 8, 2025, <https://github.com/anchore/grype/blob/main/Dockerfile>
180. jupyter/datascience-notebook - Docker Image | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/r/anchore/grype/tags>
181. anchore/anchore-engine - Docker Image | Docker Hub, accessed May 8, 2025, <https://hub.docker.com/r/anchore/anchore-engine/>
182. Image Layer Details - anchore/grype:v0.86.1 - Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/anchore/grype/v0.86.1/images/sha256-a4d05e4fa1b423996ad44fb3f29fb018672092ed9beeb63eac1b0f37c40a443d>
183. grype - overview - Chainguard Containers, accessed May 8, 2025, <https://images.chainguard.dev/directory/image/grype/overview>
184. Chainguard Containers - grype-fips - overview, accessed May 8, 2025, <https://images.chainguard.dev/directory/image/grype-fips/overview>
185. licensefinder/license\_finder - Docker Image, accessed May 8, 2025, <https://hub.docker.com/r/licensefinder/license_finder/>
186. Image Layer Details - tabll/license-finder:latest - Docker Hub, accessed May 8, 2025, <https://hub.docker.com/layers/tabll/license-finder/latest/images/sha256-6e5496ca16a689a4a4a019f640e59291c5200b7d3349d60dc45f6caf9cf4d15b?context=explore>
187. bandit configuration in MegaLinter, accessed May 8, 2025, <https://megalinter.io/v8/descriptors/python_bandit/>
188. grype configuration in MegaLinter, accessed May 8, 2025, <https://megalinter.io/beta/descriptors/repository_grype/>
189. Semgrep CE in CI, accessed May 8, 2025, <https://semgrep.dev/docs/deployment/oss-deployment>
190. semgrep configuration in MegaLinter, accessed May 8, 2025, <https://megalinter.io/8/descriptors/repository_semgrep/>
191. dockerfile ruleset - Semgrep, accessed May 8, 2025, <https://semgrep.dev/p/dockerfile>
192. syft/Dockerfile at main · anchore/syft - GitHub, accessed May 8, 2025, <https://github.com/anchore/syft/blob/main/Dockerfile>
193. Deployment Guide - PySyft - OpenMined, accessed May 8, 2025, <https://docs.openmined.org/en/latest/deployment/deployment-doc-1-2-intro-req.html>
194. A Semgrep scan is having a problem - what next?, accessed May 8, 2025, <https://semgrep.dev/docs/kb/semgrep-code/semgrep-scan-troubleshooting>
195. Configure blocking findings - Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/semgrep-ci/configuring-blocking-and-errors-in-ci>
196. Configuration — Bandit documentation - Read the Docs, accessed May 8, 2025, <https://bandit.readthedocs.io/en/latest/config.html>
197. Denormalized vs. Normalized Data - Pure Storage Blog, accessed May 8, 2025, <https://blog.purestorage.com/purely-educational/denormalized-vs-normalized-data/>
198. CodeQL CLI SARIF output - GitHub Docs, accessed May 8, 2025, <https://docs.github.com/en/code-security/codeql-cli/using-the-advanced-functionality-of-the-codeql-cli/sarif-output>
199. SARIF support for code scanning - GitHub Docs, accessed May 8, 2025, <https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning#providing-data-to-track-code-scanning-alerts-across-runs>
200. SARIF support for code scanning - GitHub Docs, accessed May 8, 2025, <https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning#sarif-specification>
201. Local CLI scans - Semgrep, accessed May 8, 2025, <https://semgrep.dev/docs/getting-started/cli-oss>
202. accessed December 31, 1969, <https://semgrep.dev/docs/security-reports/sarif-output/>
203. Test | Snyk User Docs - Snyk Documentation, accessed May 8, 2025, <https://docs.snyk.io/snyk-cli/commands/test#json-file-output---json-file-outputfilepath>
204. accessed December 31, 1969, <https://docs.snyk.io/scan-with-snyk/snyk-code/cli-for-snyk-code#sarif-output-beta>
205. accessed December 31, 1969, <https://docs.snyk.io/scan-with-snyk/snyk-open-source/cli-for-snyk-open-source#sarif-snyk-open-source-output-beta>
206. accessed December 31, 1969, <https://docs.snyk.io/snyk-api-and-custom-integrations/snyk-api/export-project-data-via-api/export-data-in-sarif-format>
207. Code test | Snyk User Docs - Snyk Documentation, accessed May 8, 2025, <https://docs.snyk.io/snyk-cli/commands/code-test#sarif-beta>
208. Test | Snyk User Docs - Snyk Documentation, accessed May 8, 2025, <https://docs.snyk.io/snyk-cli/commands/test#sarif-beta>
209. checkmarx-ts/cxone-sarif: A SARIF v2.1.0 standard compliant log generator for CheckmarxOne. - GitHub, accessed May 8, 2025, <https://github.com/checkmarx-ts/cxone-sarif>
210. accessed December 31, 1969, <https://docs.checkmarx.com/cxsca/latest/integrations/output-and-reports/generating-reports#UUID-29c643f1-e760-06f4-701f-3c5d799214f0>
211. accessed December 31, 1969, <https://docs.checkmarx.com/cxsast/latest/integrations/output-and-reports/generating-reports#UUID-29c643f1-e760-06f4-701f-3c5d799214f0>
212. accessed December 31, 1969, <https://docs.checkmarx.com/cxsast/latest/api-and-cli/checkmarx-one-cli-quick-start-guide#reporting-results>
213. accessed December 31, 1969, <https://docs.checkmarx.com/cxsca/latest/integrations/integrating-with-ci-cd-tools/checkmarx-sca-resolver#output-formats>
214. veracode/veracode-pipeline-scan-results-to-sarif - GitHub, accessed May 8, 2025, <https://github.com/veracode/veracode-pipeline-scan-results-to-sarif>
215. accessed December 31, 1969, <https://docs.veracode.com/r/c_sarif_intro>
216. accessed December 31, 1969, <https://docs.veracode.com/r/Veracode_Reporting_Overview>
217. accessed December 31, 1969, <https://docs.veracode.com/r/Veracode_SARIF_APIs>
218. accessed December 31, 1969, <https://spotbugs.github.io/spotbugs-ide/spotbugs_sarif.html>
219. Actions · GitHub Marketplace - SpotBugs with FindSecBugs, accessed May 8, 2025, <https://github.com/marketplace/actions/spotbugs-with-findsecbugs>
220. spotbugs/spotbugs-maven-plugin: Maven Mojo Plug-In to ... - GitHub, accessed May 8, 2025, <https://github.com/spotbugs/spotbugs-maven-plugin#spotbugs-goal>
221. accessed December 31, 1969, [https://github.com/spotbugs/spotbugs/issues?q=is%3Aissue+sarif](https://github.com/spotbugs/spotbugs/issues?q=is:issue+sarif)
222. dependency-check-maven - GitHub Pages, accessed May 8, 2025, <https://jeremylong.github.io/DependencyCheck/dependency-check-maven/check-mojo.html>
223. accessed December 31, 1969, <https://owasp.org/www-project-dependency-check/latest/cli/arguments.html>
224. cov-format-sarif-for-github.js - Black Duck Documentation Portal, accessed May 8, 2025, <https://documentation.blackduck.com/bundle/coverity-docs/page/commands/topics/cov-format-sarif-for-github.js.html>
225. accessed December 31, 1969, [https://github.com/trufflesecurity/trufflehog/issues?q=is%3Aissue+sarif](https://github.com/trufflesecurity/trufflehog/issues?q=is:issue+sarif)
226. Output - Trunk Docs, accessed May 8, 2025, <https://docs.trunk.io/references/cli/configuration/lint/output>
227. accessed December 31, 1969, <https://jfrog.com/help/r/jfrog-platform-rest-api/export-component-details-sarif-scit>
228. accessed December 31, 1969, <https://community.synopsys.com/s/article/Black-Duck-SARIF-Format-Support-for-GitHub-Code-Scanning>
229. accessed December 31, 1969, <https://community.synopsys.com/s/article/Outputting-Scan-Results-in-SARIF-Format-from-Black-Duck>
230. accessed December 31, 1969, <https://jfrog.com/help/r/jfrog-security-documentation/exporting-security-reports#ExportingSecurityReports-SARIFFormat>
231. accessed December 31, 1969, <https://docs.gitlab.com/ee/user/application_security/common_report_formats.html>
232. Common Vulnerability Scoring System (CVSS) Value - Ivanti, accessed May 8, 2025, <https://www.ivanti.com/blog/common-vulnerability-scoring-system-cvss>
233. accessed December 31, 1969, <https://www.sans.org/blog/a-guide-to-cvss-and-how-to-use-it-effectively/>
234. Severity Levels for Security Issues | Atlassian, accessed May 8, 2025, <https://www.atlassian.com/trust/security/security-severity-levels>
235. What is the Common Vulnerability Scoring System (CVSS)? - Balbix, accessed May 8, 2025, <https://www.balbix.com/insights/understanding-cvss-scores/>
236. Assess severity of a vulnerability - DigiCert documentation, accessed May 8, 2025, <https://docs.digicert.com/en/software-trust-manager/threat-detection/best-practices-for-common-vulnerabilities-and-exposures/assess-the-risk-of-a-vulnerability.html>
237. Guard your Codebase: Practical Steps and Tools to Prevent Malicious Code - Apiiro, accessed May 8, 2025, <https://apiiro.com/blog/guard-your-codebase-practical-steps-and-tools-to-prevent-malicious-code/>
238. Probable false positive from Findbugs? - java - Stack Overflow, accessed May 8, 2025, <https://stackoverflow.com/questions/15547475/probable-false-positive-from-findbugs>
239. Contextual Risk Estimation for Effective Prioritization | Microsoft ..., accessed May 8, 2025, <https://techcommunity.microsoft.com/t5/microsoft-defender-for-cloud/contextual-risk-estimation-for-effective-prioritization/ba-p/4035425>
240. Using Reachability & Exploitability to Reduce CVE False Positives - OpsMx, accessed May 8, 2025, <https://www.opsmx.com/blog/reducing-cve-false-positives-using-reachability-and-exploitability/>
241. bandit 1.6.1 - PyPI, accessed May 8, 2025, <https://pypi.org/project/bandit/1.6.1/>
242. Bandit Documentation, accessed May 8, 2025, <https://bandit.readthedocs.io/_/downloads/en/1.7.3/pdf/>
243. View github: anchore/grype | OpenText Core SCA - Debricked, accessed May 8, 2025, <https://debricked.com/select/package/github-anchore/grype>
244. Manage Duplicate Findings - DefectDojo Documentation, accessed May 8, 2025, <https://docs.defectdojo.com/en/working_with_findings/findings_workflows/manage_duplicate_findings/>
245. Charlotte AI: Generative AI for Cybersecurity | CrowdStrike, accessed May 8, 2025, <https://www.crowdstrike.com/en-us/platform/charlotte-ai/>
246. Review | AquilaX Docs, accessed May 8, 2025, <https://docs.aquilax.ai/ai-models/review>
247. What Is Alert Triage in Cybersecurity? Steps, Challenges, and Tips | Corelight, accessed May 8, 2025, <https://corelight.com/resources/glossary/alert-triage>
248. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/8935301>
249. accessed December 31, 1969, <https://www.recordedfuture.com/blog/closing-the-loop-using-feedback-to-improve-security-alerting>
250. Confidence Scoring in Threat Intelligence | Cyware, accessed May 8, 2025, <https://www.cyware.com/resources/security-guides/cyber-threat-intelligence/what-is-confidence-scoring-in-threat-intelligence>
251. What is the vulnerability confidence rating and why is it important ..., accessed May 8, 2025, <https://www.acunetix.com/support/docs/faqs/what-is-vulnerability-confidence-rating/>
252. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9191161>
253. accessed December 31, 1969, <https://www.researchgate.net/publication/334890090_Confidence_Levels_in_Static_Analysis_Tools_A_Case_Study_with_FindBugs_and_SpotBugs>
254. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/8418609>
255. accessed December 31, 1969, <https://www.researchgate.net/publication/344061289_A_Methodology_for_Evaluating_and_Comparing_Static_Application_Security_Testing_SAST_Tools_Effectiveness_in_Detecting_Vulnerabilities_and_Reporting_False_PositivesNegatives>
256. Source Code Analysis Tools | OWASP Foundation, accessed May 8, 2025, <https://owasp.org/www-community/Source_Code_Analysis_Tools#Assessing_Confidence>
257. accessed December 31, 1969, <https://www.veracode.com/blog/security-news/how-flaw-matching-and-machine-learning-reduce-false-positives>
258. accessed December 31, 1969, <https://www.codacy.com/blog/how-to-prioritize-and-fix-static-analysis-issues/>
259. accessed December 31, 1969, <https://www.elastic.co/blog/machine-learning-categorization-nlp-elasticsearch>
260. Digital Fingerprinting in Cybersecurity: OS, Nmap, & More, accessed May 8, 2025, <https://www.bitsight.com/learn/digital-fingerprinting>
261. Fingerprinting Vulnerabilities - Veracode, accessed May 8, 2025, <https://www.veracode.com/security/fingerprinting-vulnerabilities>
262. What is Fingerprinting in Cybersecurity? - Recorded Future, accessed May 8, 2025, <https://www.recordedfuture.com/threat-intelligence-101/vulnerability-management-threat-hunting/fingerprinting-in-cybersecurity>
263. Fingerprint (computing) - Wikipedia, accessed May 8, 2025, <https://en.wikipedia.org/wiki/Fingerprint_(computing)>
264. Fingerprint-Based Data Deduplication Using a Mathematical Bounded Linear Hash Function, accessed May 8, 2025, <https://www.mdpi.com/2073-8994/13/11/1978>
265. Hash Value: The Fingerprint of Data: Using Hash Values to Verify Digital Evidence, accessed May 8, 2025, <https://fastercapital.com/content/Hash-Value--The-Fingerprint-of-Data--Using-Hash-Values-to-Verify-Digital-Evidence.html>
266. FED: Fast and Efficient Dataset Deduplication Framework with GPU Acceleration - arXiv, accessed May 8, 2025, <https://arxiv.org/html/2501.01046v1>
267. accessed December 31, 1969, <https://support.sonatype.com/hc/en-us/articles/360051364554-Understanding-how-Sonatype-Lifecycle-handles-duplicate-vulnerabilities>
268. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9463095>
269. accessed December 31, 1969, <https://www.researchgate.net/publication/344560089_VulDedupe_A_Vulnerability_Deduplication_System_for_Open-Source_Software_Supply_Chains>
270. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9327640>
271. accessed December 31, 1969, <https://www.researchgate.net/publication/332322491_Fingerprinting_Software_Vulnerabilities_A_Systematic_Mapping_Study>
272. accessed December 31, 1969, <https://resources.contrastsecurity.com/hubfs/Understanding_Vulnerability_Fingerprinting_A_Developers_Guide_to_Deduplication_and_Tracking.pdf>
273. accessed December 31, 1969, <https://security.googleblog.com/2023/09/how-osv-improves-vulnerability.html>
274. Include fingerprint in SARIF output · Issue #1058 - GitHub, accessed May 8, 2025, <https://github.com/hadolint/hadolint/issues/1058>
275. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9607771>
276. accessed December 31, 1969, <https://github.com/microsoft/sarif-sdk/issues/1875>
277. Deep learning for blocking in entity matching: a design space exploration - ResearchGate, accessed May 8, 2025, <https://www.researchgate.net/publication/355693740_Deep_learning_for_blocking_in_entity_matching_a_design_space_exploration>
278. How Data Deduplication Improves Investigations? - Data Ladder, accessed May 8, 2025, <https://dataladder.com/how-data-deduplication-improves-investigations/?imz_s=sbaksln69j940ble6eatuom730/>
279. Safe C++, accessed May 8, 2025, <https://safecpp.org/>
280. The 2025 In-Depth Guide to OWASP Top 10 Vulnerabilities & How to Prevent Them - Jit.io, accessed May 8, 2025, <https://www.jit.io/resources/security-standards/the-in-depth-guide-to-owasps-top-10-vulnerabilities>
281. accessed December 31, 1969, <https://www.recordedfuture.com/blog/automating-vulnerability-management>
282. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9796575>
283. accessed December 31, 1969, <https://www.securitybridge.com/news/the-art-of-deduplication-how-to-effectively-manage-security-vulnerabilities.html>
284. accessed December 31, 1969, <https://checkmarx.com/blog/application-security-managing-sca-risks-part-2-prioritization-and-remediation/>
285. Coordinating Defense:, accessed May 8, 2025, <https://owasp.org/www-chapter-sofia/assets/presentations/202503%20-%20Coordinating%20Defense:%20My%20take%20on%20effective%20security%20testing%20strategies%20by%20Dimo%20Stankov.pdf>
286. AI Use Case Inventory | Homeland Security, accessed May 8, 2025, <https://www.dhs.gov/archive/data/AI_inventory>
287. fdtn-ai/Foundation-Sec-8B - Hugging Face, accessed May 8, 2025, <https://huggingface.co/fdtn-ai/Foundation-Sec-8B>
288. A SURVEY ON THE USE OF DATA CLUSTERING FOR INTRUSION DETECTION SYSTEM IN CYBERSECURITY - PMC, accessed May 8, 2025, <https://pmc.ncbi.nlm.nih.gov/articles/PMC8289996/>
289. (PDF) The Role of Natural Language Processing in Cybersecurity - ResearchGate, accessed May 8, 2025, <https://www.researchgate.net/publication/387252722_The_Role_of_Natural_Language_Processing_in_Cybersecurity>
290. www.aclweb.org, accessed May 8, 2025, <https://www.aclweb.org/anthology/2020.lrec-1.308.pdf>
291. accessed December 31, 1969, <https://ieeexplore.ieee.org/document/9796575>
292. arxiv.org, accessed May 8, 2025, <https://arxiv.org/abs/2109.07661>
293. (PDF) AI-Driven Categorization and Deduplication - ResearchGate, accessed May 8, 2025, <https://www.researchgate.net/publication/389210560_AI-Driven_Categorization_and_Deduplication>
294. Reduce Noise With Alert Deduplication - ilert, accessed May 8, 2025, <https://www.ilert.com/ai-incident-management-guide/reduce-noise-with-alert-deduplication>
295. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9040590>
296. accessed December 31, 1969, <https://www.researchgate.net/publication/348500944_An_Empirical_Study_on_the_Use_of_Machine_Learning_for_Vulnerability_Report_Deduplication>
297. accessed December 31, 1969, <https://blog.sonarsource.com/accurate-issue-tracking-for-static-analysis/>
298. accessed December 31, 1969, <https://engineering.grammarly.com/blog/managing-code-locations-at-scale-with-asts-and-other-techniques/>
299. VulMatch: Binary-Level Vulnerability Detection Through Signature - ResearchGate, accessed May 8, 2025, <https://www.researchgate.net/publication/389844102_VulMatch_Binary-Level_Vulnerability_Detection_Through_Signature>
300. TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities, accessed May 8, 2025, <https://www.researchgate.net/publication/365230735_TRACER_Signature-based_Static_Analysis_for_Detecting_Recurring_Vulnerabilities>
301. (PDF) Generalized Vulnerability Extrapolation using Abstract Syntax Trees - ResearchGate, accessed May 8, 2025, <https://www.researchgate.net/publication/233997025_Generalized_Vulnerability_Extrapolation_using_Abstract_Syntax_Trees>
302. Generalized Vulnerability Extrapolation using Abstract Syntax Trees - MLSEC, accessed May 8, 2025, <https://mlsec.tu-berlin.de/docs/2012-acsac.pdf>
303. Vulnerability Detection in Source Code via Subgraph Isomorphism Analysis, accessed May 8, 2025, <https://wpcdn.web.wsu.edu/wp-vcea/uploads/sites/3267/2024/05/Vulnerability-Detection-in-Source-Code-via-Subgraph-Isomorphism-Analysis.pdf>
304. General | Developer Tools | Abstract Syntax Tree - Codecademy, accessed May 8, 2025, <https://www.codecademy.com/resources/docs/general/developer-tools/abstract-syntax-tree>
305. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/8632589>
306. accessed December 31, 1969, <https://www.researchgate.net/publication/320293850_AST-Based_Technique_for_Tracking_Static_Analysis_Alerts_Through_Code_Transformations>
307. accessed December 31, 1969, <https://www.researchgate.net/publication/351648095_A_Robust_Approach_for_Tracking_Static_Analysis_Warnings_Across_Software_Versions>
308. accessed December 31, 1969, <https://dl.acm.org/doi/pdf/10.1145/3337970.3337971>
309. Implementing: Storing and Securing Your Data - Research Data ..., accessed May 8, 2025, <https://guides.lib.uw.edu/research/dmg/storage-security>
310. Secure Data Storage: The 6 Best Practices | Object First, accessed May 8, 2025, <https://objectfirst.com/guides/data-storage/secure-data-storage/>
311. Can Neo4j Replace PostgreSQL in Healthcare? - PMC, accessed May 8, 2025, <https://pmc.ncbi.nlm.nih.gov/articles/PMC7233060/>
312. Compare Elasticsearch vs PostgreSQL - InfluxDB, accessed May 8, 2025, <https://www.influxdata.com/comparison/elasticsearch-vs-postgres/>
313. Elasticsearch vs Graph Database | e-Zest, accessed May 8, 2025, <https://blog.e-zest.com/elasticsearch-vs-graph-database/>
314. Tune for search speed | Elastic Docs, accessed May 8, 2025, <https://www.elastic.co/guide/en/elasticsearch/reference/current/tune-for-search-speed.html>
315. Vector database vs. graph database: Understanding the differences | Elastic Blog, accessed May 8, 2025, <https://www.elastic.co/blog/vector-database-vs-graph-database>
316. Elasticsearch vs Neo4j on Vector Search Capabilities - Zilliz blog, accessed May 8, 2025, <https://zilliz.com/blog/elasticsearch-vs-neo4j-a-comprehensive-vector-database-comparison>
317. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9424198>
318. accessed December 31, 1969, <https://www.researchgate.net/publication/349181090_CyGraph_A_Graph-Based_Cybersecurity_Data_Analysis_Platform>
319. accessed December 31, 1969, <https://www.researchgate.net/publication/361010992_A_Graph-Based_Approach_for_Cybersecurity_Knowledge_Graph_Construction_and_Analysis>
320. accessed December 31, 1969, <https://medium.com/neo4j/graph-databases-for-cybersecurity-enhancing-threat-detection-and-response-9a727c73cf9c>
321. accessed December 31, 1969, <https://neo4j.com/blog/knowledge-graphs-cybersecurity-graphcipher/>
322. Documentation: 17: CREATE INDEX - PostgreSQL, accessed May 8, 2025, <https://www.postgresql.org/docs/current/sql-createindex.html>
323. Optimal Indexing Strategies for PostgreSQL: Boosting Performance and Scalability, accessed May 8, 2025, <https://speakerdeck.com/shiviyer007/optimal-indexing-strategies-for-postgresql-boosting-performance-and-scalability>
324. PostgreSQL Indexing Best Practices Guide - MyDBOPS, accessed May 8, 2025, <https://www.mydbops.com/blog/postgresql-indexing-best-practices-guide>
325. Mapping | Elastic Docs, accessed May 8, 2025, <https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html>
326. Update Security index settings | Elasticsearch Guide [8.18] | Elastic, accessed May 8, 2025, <https://www.elastic.co/guide/en/elasticsearch/reference/8.18/security-api-update-settings.html>
327. Get Security index settings | Elasticsearch Guide [8.18] | Elastic, accessed May 8, 2025, <https://www.elastic.co/guide/en/elasticsearch/reference/8.18/security-api-get-settings.html>
328. OWASP Juice Shop | OWASP Foundation, accessed May 8, 2025, <https://owasp.org/www-project-juice-shop/>
329. accessed December 31, 1969, <https://owasp.org/www-project-nodegoat/>
330. Pygoat | OWASP Foundation, accessed May 8, 2025, <https://owasp.org/www-project-pygoat/>
331. OWASP WebGoat | OWASP Foundation, accessed May 8, 2025, <https://owasp.org/www-project-webgoat/>
332. accessed December 31, 1969, <https://owasp.org/www-project-security-knowledge-framework/>
333. Projects - OWASP Foundation, accessed May 8, 2025, <https://owasp.org/projects/>
334. guizmaii/akka-samples-cluster-scala - GitHub, accessed May 8, 2025, <https://github.com/guizmaii/akka-samples-cluster-scala>
335. playframework/play-scala-starter-example - GitHub, accessed May 8, 2025, <https://github.com/playframework/play-scala-starter-example>
336. Exploit Prediction Scoring System (EPSS), accessed May 8, 2025, <https://www.first.org/epss/api>
337. The EPSS Model, accessed May 8, 2025, <https://www.first.org/epss/model>
338. EPSS Scores in the GitHub Advisory Database - GitHub Changelog, accessed May 8, 2025, <https://github.blog/changelog/2024-10-10-epss-scores-in-the-github-advisory-database/>
339. Reducing the Significant Risk of Known Exploited Vulnerabilities - CISA, accessed May 8, 2025, <https://www.cisa.gov/known-exploited-vulnerabilities>
340. Known Exploited Vulnerabilities Catalog | CISA, accessed May 8, 2025, <https://www.cisa.gov/known-exploited-vulnerabilities-catalog>
341. accessed December 31, 1969, <https://ieeexplore.ieee.org/abstract/document/9714431>
342. accessed December 31, 1969, <https://docs.gitlab.com/ee/user/project/code_owners/>
343. git-blame Documentation - Git, accessed May 8, 2025, <https://git-scm.com/docs/git-blame>
344. CyberSecurity Asset Management 3.0 (CSAM) | Qualys, accessed May 8, 2025, <https://www.qualys.com/apps/cybersecurity-asset-management/>
345. Leveraging Data Enrichment in Cyber Threat Intelligence (CTI ..., accessed May 8, 2025, <https://www.anomali.com/blog/leveraging-data-enrichment-in-threat-intelligence>
346. OWASP Developer Guide | DefectDojo, accessed May 8, 2025, <https://owasp.org/www-project-developer-guide/draft/verification/vulnerability_management/defectdojo/>
347. DefectDojo API v2, accessed May 8, 2025, <https://docs.defectdojo.com/en/api/api-v2-docs/>
348. Finding Workflows & Status - DefectDojo Documentation, accessed May 8, 2025, <https://docs.defectdojo.com/en/working_with_findings/findings_workflows/>
349. OWASP Defectdojo – OWASP Nest, accessed May 8, 2025, <https://nest.owasp.org/projects/defectdojo>
350. About Our Documentation - DefectDojo Documentation, accessed May 8, 2025, <https://docs.defectdojo.com/en/about_defectdojo/about_docs/>
351. DefectDojo - DefectDojo - DefectDojo, accessed May 8, 2025, <https://www.defectdojo.com/>
352. accessed December 31, 1969, <https://defectdojo.github.io/django-DefectDojo/integrations/importing_via_the_api/>
353. accessed December 31, 1969, <https://defectdojo.github.io/django-DefectDojo/integrations/parsers/>
354. DefectDojo Documentation | DefectDojo Documentation, accessed May 8, 2025, <https://documentation.defectdojo.com/>
355. Integrating Faraday Bags into Corporate Security Protocols: Preventing Wireless Leaks in Sensitive Environments - Jemic Shielding Technology, accessed May 8, 2025, <https://www.jemic.com/faraday-bag/faraday-bags-corporate-security-protocols-wireless-leaks-sensitive-environments/>
356. The Ultimate Guide to Static Code Analysis in 2025 + 14 SCA Tools - CodeAnt AI, accessed May 8, 2025, <https://www.codeant.ai/blogs/static-code-analysis-tools>
357. Sign Container Images With Cosign - Exoscale, accessed May 8, 2025, <https://www.exoscale.com/syslog/2025-02-24-sign-container-images-with-cosign/>
358. Sigstore Quickstart with Cosign, accessed May 8, 2025, <https://docs.sigstore.dev/quickstart/quickstart-cosign/>
359. engineerd/in-toto-container: Programmatically running in-toto verifications in a container - GitHub, accessed May 8, 2025, <https://github.com/engineerd/in-toto-container>
360. Documentation | in-toto, accessed May 8, 2025, <https://in-toto.io/docs/>